FiPy
User's Guide

Daniel Wheeler
Jonathan E. Guyer
James A. Warren

Metallurgy Division
and the Center for Theoretical and Computational Materials Science
Materials Science and Engineering Laboratory

June 6, 2006

Version 1.1

Contents

Part I

Introduction

Introduction Contents

Chapter 1

Overview

FiPy is an object oriented, partial differential equation (PDE) solver, written in Python [1], based on a standard finite volume (FV) approach. The framework has been developed in the Metallurgy Division and Center for Theoretical and Computational Materials Science (CTCMS), in the Materials Science and Engineering Laboratory (MSEL) at the National Institute of Standards and Technology (NIST).

The solution of coupled sets of PDEs is ubiquitous to the numerical simulation of science problems. Numerous PDE solvers exist, using a variety of languages and numerical approaches. Many are proprietary, expensive and difficult to customize. As a result, scientists spend considerable resources repeatedly developing limited tools for specific problems. Our approach, combining the FV method and Python, provides a tool that is extensible, powerful and freely available. A significant advantage to Python is the existing suite of tools for array calculations, sparse matrices and data rendering.

The FiPy framework includes terms for transient diffusion, convection and standard sources, enabling the solution of arbitrary combinations of coupled elliptic, hyperbolic and parabolic PDEs. Currently implemented models include phase field [2, 3, 4] treatments of polycrystalline, dendritic, and electrochemical phase transformations as well as a level set treatment of the electrodeposition process [5].

The latest information about FiPy can be found at http://www.ctcms.nist.gov/fipy/.

1.1 Even if you don't read manuals...

...please read Chapter 2 "Installation and Usage" and Chapter 6 "Frequently Asked Questions".

1.2 What's new in version 1.1

The significant changes since version 1.0 are:

- Memory efficiency has been improved in a number of ways, but most significantly by:

    - not caching all intermediate `Variable` values.
    - introducing `UniformGrid` classes that calculate geometric arrays on the fly.

    Details of these improvements are presented in Chapter 5 "Efficiency".

- Installation on Windows has been made considerably easier by constructing executable installers for FiPy and its dependencies. Instructions for Windows installation can be found in Section 2.7 "Windows Installation".

- The arithmetic for `Variable` subclasses now works, and returns sensible answers. For example, `VectorCellVariable * CellVariable` returns a `VectorCellVariable`.

- `PeriodicGrid` meshes have been implemented. Currently, however, there and no examples of their use in the manual.

- Many of the examples have been completely rewritten

    - A basic 1D diffusion problem now serves as a general tutorial for setting up any problem in FiPy.
    - Several more phase field examples have been added that should make it clearer how to get from the simple 1D case to the more elaborate multicomponent, multidimensional, and anisotropic models.
    - The "Superfill" examples have been substantially improved with better functionality and documentation.
    - An example of fluid flow with the classic Stokes moving lid has been added.

- A clear distinction has been made between solving an equation via *solve()* and iterating an non-linear equation to solution via *sweep()*. An extensive explanation of the concepts involved has been added to Chapter 6 "Frequently Asked Questions".

- Added a *MultiViewer* class that automatically groups several viewers together if the variables couldn't be displayed by a single viewer.

- The abbreviated syntax `from fipy import Class` or `from fipy import *` promised in version 1.0 actually works now. The examples all still use the fully qualified names.

- The repository has been converted from a CVS to a Subversion repository. Details on how to check out the new repository are given in the Chapter 2 "Installation and Usage".

- The FiPy repository has also been moved from Sourceforge to the Materials Digital Library Pathway.

1.3   Download and Installation

Please refer to Chapter 2 "Installation and Usage" for details on download and installation. FiPy can be redistributed and/or modified freely, provided that any derivative works bear some notice that they are derived from it, and any modified versions bear some notice that they have been modified.

1.4   Support

You can communicate with the FiPy developers and with other users via our mailing list and we welcome you to use the tracking system for bugs, support requests, feature requests and patch submissions [6, 7]. We welcome collaborative efforts on this project.

1.5   Conventions and Notation

FiPy is driven by Python script files than you can view or modify in any text editor. FiPy sessions are invoked from a command-line shell, such as `tcsh` or `bash`.
Throughout, text to be typed at the keyboard will appear `like this`. Commands to be issued from an interactive shell will appear:

        $ like this

where you would enter the text ("`like this`") following the shell prompt, denoted by "`$`".
Text blocks of the form:

```
>>> a = 3 * 4
>>> a
12
>>> if a == 12:
...     print "a is twelve"
...
a is twelve
```

are intended to indicate an interactive session in the Python interpreter. We will refer to these as "interactive sessions" or as "doctest blocks". The text ">>>" at the beginning of a line denotes the *primary prompt*, calling for input of a Python command. The text "..." denotes the *secondary prompt*, which calls for input that continues from the line above, when required by Python syntax. All remaining lines, which begin at the left margin, denote output from the Python interpreter. In all cases, the prompt is supplied by the Python interpreter and should not be typed by you.

Warning

Python is sensitive to indentation and care should be taken to enter text exactly as it appears in the examples.

When references are made to file system paths, it is assumed that the current working directory is the FiPy distribution directory, refered to as the "base directory", such that:

```
examples/diffusion/steadyState/mesh1D/input.py
```

will correspond to, *e.g.*:

```
/some/where/FiPy-1.0/examples/diffusion/steadyState/mesh1D/input.py
```

Paths will always be rendered using POSIX conventions (path elements separated by "/"). Any references of the form:

```
examples.diffusion.steadyState.mesh1D.input
```

are in the Python module notation and correspond to the equivalent POSIX path given above.
We may at times use a

Note

to indicate something that may be of interest

or a

Warning

to indicate something that could cause serious problems.

1.6   Mailing List

In order to discuss FiPy with other users and with the developers, we encourage you to sign up for the mailing list by sending a subscription email:

To: `listproc@nist.gov`

Subject: *(optional)*

Body: `subscribe fipy` *Your Name*

Once you are subscribed, you can post messages to the list simply by addressing email to mailto:fipy@nist.gov. If you are new to mailing lists, you may want to read the following resource about asking effective questions: http://www.catb.org/~esr/faqs/smart-questions.html

List Archive

http://dir.gmane.org/gmane.comp.python.fipy

The mailing list archive is hosted by GMANE. Any mail sent to fipy@nist.gov will appear in this publicly available archive.

Chapter 2

Installation and Usage

The FiPy finite volume PDE solver relies on several third-party packages. It is *best to obtain and install those first*, before attempting to install FiPy.

Note

Most of the installation steps will involve a variant on the command:

    $ python setup.py ...

In addition to the specific commands given here, further information about each setup.py script is available by typing:

    $ python setup.py --help

For each package, please follow any instructions given in its *README* or *INSTALLATION* files.

2.1   Privileges

If you do not have administrative privileges on your computer, or if for any reason you don't want to tamper with your existing Python installation, most packages (including FiPy) will allow you to install to an alternate location. Instead of installing these packages with `python setup.py install`, you would use `python setup.py install --home=<dir>`, where `<dir>` is the desired installation directory (usually "~" to indicate your home directory). You will then need to append `<dir>/lib/python` to your PYTHONPATH environment variable. See the Alternate Installation section of the Python document "Installing Python Modules" [8] for more information, such as circumstances in which you should use `--prefix` instead of `--home`.

2.2   Prerequisites

Operating System

FiPy has been developed and tested on the following operating systems, Mac OS X 10.3 "Panther" & 10.4 "Tiger", Debian Linux 3.1 "Sarge", Windows XP and Windows 2000. We welcome reports of compatibility with other systems, along with any steps necessary to install.

> **Note**
>
> Simple instructions for Windows users are in Section 2.7 "Windows Installation".

The only elements of FiPy that are likely to be platform-dependent are the viewers, but at least one viewer should work on each platform. All other aspects should function on any platform that has a recent Python installation.
Many of the packages listed below have prebuilt installers for different platforms (particularly for Windows). These installers can save considerable time and effort compared to configuring and building from source, although they frequently comprise somewhat older versions of the respective code. Whether building from source or using a prebuilt installer, please read and follow explicitly any instructions given in the respective packages' `README` and `INSTALLATION` files.

Required Packages

> **Warning**
>
> FiPy will not run if the following items are not installed.

> **Attention!**
>
> On some platforms, it may be possible to satisfy most, if not all, of the following prerequisites by installing the "Enthought Edition of Python (Enthon)". We are aware of Enthon for Windows [9] and Enthon for Mac OS X [10].

Python

http://www.python.org/
FiPy is written in the Python language and requires a Python installation to run. Python comes pre-installed on many operating systems, which you can check by opening a terminal and typing `python`, *e.g.*:

```
$ python
Python 2.3 (#1, Sep 13 2003, 00:49:11)
...
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If necessary, you can download and install it for your platform [11].

> **Note**
>
> FiPy requires at least version 2.3 of Python.

Numeric

http://sourceforge.net/projects/numpy

Obtain and install the Numeric package. FiPy requires at least version 23.1 of Numeric. Numeric Windows installers are available.

> **Attention!**
>
> FiPy cannot presently use the newer numarray or NumPy packages. Due to a quirk of history, the Numeric package has been casually referred to as "numpy" for many years, but it is distinct from (although related to) NumPy.

> **Note**
>
> Because of a peculiarity in the way that Numeric is structured, the `--home=<dir>` installation option described in Privileges will not work quite as intended. To correct this problem, add `<dir>/lib/python/Numeric` to your `PYTHONPATH` environment variable.

PySparse

http://pysparse.sourceforge.net

FiPy requires Roman Geus' PySparse package.

You can download the PySparse archive or check it out via anonymous CVS download:

```
$ cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/pysparse login
```

and press enter at the password prompt, then:

```
$ cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/pysparse \
>       checkout pysparse
```

From within the `pysparse` base directory, follow its included instructions for building PySparse for your platform. PySparse Windows installers are available.

> **Note**
>
> Windows users who choose to build from source should pay particular attention to the instructions in the INSTALL file in the base PySparse directory.

> **Warning**
>
> If pysparse is installed in a local directory a further path may have to be added to the PYTHONPATH environment variable. For example, if
>
> ```
> $ python setup.py install --home=/some/directory/some/where
> ```
>
> then both /some/directory/some/where and /some/directory/some/where/lib/python are required to be added to the PYTHONPATH. e.g.
>
> ```
> $ set PYTHONPATH=/some/directory/some/where:/some/directory/some/where/lib/python
> ```

Viewers

FiPy will work perfectly well without them, but at least one of the following packages will be required to view the results of FiPy calculations. FiPy will select the first viewer that is available from the list below. If more than one is installed, specify a viewer by setting the FIPY_VIEWER environment variable to either "gist", "gnuplot" or "matplotlib".

Matplotlib

http://matplotlib.sourceforge.net
Matplotlib is a Python package that displays publication quality results. It displays both 1D X-Y type plots and 2D contour plots for structured data. It does not display unstructured 2D data or 3D data. It works on all common platforms and produces publication quality hard copies. Version 0.72.1 or higher is required. Matplotlib installers for specific platforms are available [12].

Gnuplot-py

http://gnuplot-py.sourceforge.net
Gnuplot.py is a Python package that interfaces to gnuplot, the popular open-source plotting program. It displays both 1D X-Y type plots and 2D contour plots for structured data but not for unstructured data or 3D data. It works on all common platforms and produces hard copies, however, it sometimes breaks on Windows. As a general remark, the viewing quality using either Pygist or Matplotlib is preferable.

Pygist

http://bonsai.ims.u-tokyo.ac.jp/~mdehoon/software/python/pygist.html
The Pygist package can be used to display simulation results. It displays both 1D X-Y type plots and 2D contour plots for both structured and unstructured data. It does not display 3D data. Although stated as working on Windows, it does not seem to do a good job of rendering on this platform. Pygist works fine on other common platforms.

Warning

The facility to produce hard copies in Pygist does not work very well and may crash the FiPy run. ".eps" and ".cgm" export seem to work.

Attention!

Pygist seems to be unable to make contour plots on x86_64 architectures running Linux.

Note

If you experience difficulty building the native Pygist viewer on Mac OS X, you may wish to build the package with the --x11 option described in its documentation.

Note

Pygist can have problems finding color pallets, such as "heat.gp" and "work.gs", when installed locally. A fix for this problem is to copy the required pallets into the local directory. The color pallets are in "src/gist/g" of the base Pygist installation.

> **Warning**
>
> If Pygist is installed in a local directory a further path may have to be added to the PYTHONPATH environment variable. For example, if
>
> $ python setup.py install --home=/some/directory/some/where
>
> then both /some/directory/some/where and /some/directory/some/where/lib/python/gist are required to be added to the PYTHONPATH. e.g.
>
> $ set
> PYTHONPATH=/some/directory/some/where:/some/directory/some/where/lib/python/gist

MayaVi

http://mayavi.sourceforge.net
The MayaVi Data Visualizer is a free, easy to use scientific data visualizer. It displays 1D, 2D and 3D data. It is the only FiPy viewer available for 3D data. Other viewers are probably better for 1D or 2D viewing. The Enthon for Windows version of python is recommended for Windows users who wish to use MayaVi.

> **Warning**
>
> MayaVi can only display 3D meshes consisting entirely of tetrahedrons or wedge elements. The ordering of vertices for other mesh types may not work.

> **Note**
>
> Is is also necessary to install the PyVTK package to use the FiPy MayaVi viewers.

Recommended Packages

> **Note**
>
> The following packages are not required to run FiPy, but they can be helpful.

SciPy

http://www.scipy.org/
Significantly improved performance has been achieved with the judicious use of C language inlining, via the weave module of the SciPy package. SciPy installers for specific platforms [13] are available.

> **Attention!**
>
> SciPy is undergoing a lot of development lately but, as yet, FiPy still requires "old SciPy". Be sure to install version 0.3.x of `SciPy_complete`.

> **Note**
>
> A handful of test cases use functions from the SciPy library and will throw errors if it is missing.

gmsh

http://www.geuz.org/gmsh/
It is possible to create irregular meshes with this package.

> **Note**
>
> The Mac OS X distribution of gmsh provides a nice graphical tool, but is structured a bit differently than on other platforms. To access the underlying shell tool, create a shell alias called `gmsh` that corresponds to `<Gmsh path>/Gmsh.app/Contents/MacOS/Gmsh`.

2.3   Obtaining **FiPy**

FiPy is freely available for download via Subversion or as a compressed archive [14]. To obtain FiPy via anonymous Subversion, issue the following commands:

```
$ svn checkout http://matdl-osi.org/svn/fipy/tags/CURRENT
```

This will download a fairly stable version of FiPy (somewhere between version 1.1 and HEAD). If you prefer, you can download FiPy version 1.1 (recommended) with:

```
$ svn checkout http://matdl-osi.org/svn/fipy/tags/STABLE
```

Further information about Subversion can be found in Section 2.8 and in the online Subversion Red Bean book [15].

> **Warning**
>
> Keep in mind that if you choose to download the compressed archive you will then need to preserve your changes when upgrades to FiPy become available (upgrades via Subversion will handle this issue automatically).

Manual

You can download the latest manual [16]. Alternatively, it may be possible to build a fresh copy by issuing the following command in the base directory:

```
$ python setup.py build_docs --latex --manual
```

> **Note**
>
> This mechanism is intended primarily for the developers. A command-line pdfTeX installation and several LATEX packages are required; particularly `memoir.cls`. You will also need to add `~/path/to/fipy/utils` to your `PYTHONPATH` environment variable.

2.4   Testing **FiPy**

From the base directory, you can verify that FiPy works properly by executing:

```
$ python setup.py test
```

Depending on the packages you chose to install in Recommended Packages, be sure to set the appropriate environment variables. You can expect a few errors if you did not install all of the recommended packages. If you chose to install the weave package, you should rerun the tests with:

```
$ python setup.py test --inline
```

A few tests will fail the first time as a result of the messages output in the course of caching the compiled inline code, but a repeat test should have no failures (although see "`repairing catalog by removing key`" in Chapter 6 "Frequently Asked Questions").

> **Note**
>
> In order for Python to find the FiPy modules, you will need to ensure that the base directory is added to your PYTHONPATH environment variable, *e.g.*:
>
> ```
> $ setenv PYTHONPATH .:${PYTHONPATH}
> ```
>
> or:
>
> ```
> $ export PYTHONPATH=.:${PYTHONPATH}
> ```

2.5 Installing FiPy

It is not necessary to formally install FiPy, but if you wish to do so and you are confident that all of the requisite packages have been installed properly and FiPy passes its tests, you can install it by typing:

```
$ python setup.py install
```

at the command line. Alternatively, you may choose not to formally install FiPy and to simply work within the base directory instead.

If you choose to install, Python will find your FiPy modules automatically. If you choose not to install, then you will need to ensure that the FiPy distribution directory is appended to your PYTHONPATH environment variable (either "." if you are working within the FiPy directory, or "~/path/to/fipy" if you are working in your own directory).

2.6 Using FiPy

To see examples of problems that FiPy is capable of solving, you can run any of the scripts in Part II. You can type, *e.g.*:

```
$ python examples/diffusion/steadyState/mesh1D/input.py
```

at the command line, which should produce a graphical display of the solution.

With judicious use of the weave package, we have been able to obtain significantly improved performance for some problems, while keeping the code as clear as possible. You can invoke this faster code by passing the `--inline` option at the command line, *i.e.*:

```
$ python examples/diffusion/steadyState/mesh1D/input.py --inline
```

Some modest speed efficiency gains can be made with the use of the `--cache` flag. This flag instructs FiPy to trade memory for speed. This flag can be invoked with, *e.g.*:

```
$ python examples/diffusion/steadyState/mesh1D/input.py --inline --cache
```

In order to customize the examples, or to develop your own scripts, some knowledge of Python syntax is required. We recommend you familiarize yourself with the excellent Python tutorial [17] or with Dive Into Python [18].

2.7 Windows Installation

These instructions are simple step by step instructions for the Windows XP and Windows 2000 platforms. Please see the more general Chapter 2 "Installation and Usage" for detailed installation instructions. These step by step instructions are not the only way to set up FiPy on a Windows OS but represent the most expedient way from our experience to have a usable installation up and running.

Note

These steps require downloading installation `.exe` binary files from various web sites. When given the choice to `run` (`open`) or `save` these files it is best to simply choose `run` (`open`).

Step 1: Install Python

http://download.enthought.com/enthought_python-2.3.5-1076-wx26-n238.exe
Download and run the Enthought Edition installer. Open the `C:\Python23\Lib\site-packages\vtk_python\` directory in a browser and move `vtkRenderingPythonTkWidgets.dll` to the `C:\Python23\` directory.

Step 2: Install PyVTK

http://www.ctcms.nist.gov/fipy/download/PyVTK-0.4.70.win32.exe
Download and run the PyVTK installer.

Step 3: Install Pysparse

http://prdownloads.sourceforge.net/pysparse/pysparse-0.34.032.win32-py2.3.exe?download
Download and run the Pysparse installer.

Step 4: Install FiPy

http://www.ctcms.nist.gov/fipy/download/FiPy-1.1.win32.zip
Download and unpack the zip file. Run the FiPy installer `FiPy-1.1.win32.exe`, which should be in the base FiPy-1.1 directory.

Step 5: Install Matplotlib

http://prdownloads.sourceforge.net/matplotlib/matplotlib-0.87.2.win32-py2.3.exe?download
Download and run the Matplotlib installer.

Step 6: Install Gmsh

http://www.geuz.org/gmsh/bin/Windows/gmsh-1.65.0-Windows.zip
Download and unpack Gmsh for Windows. Open the unpacked folder with a browser and copy the `gmsh.exe` file to `C:\Python23`.

Step 7: Run IDLE

The IDLE environment is recommended for running FiPy. In order to have interactive viewing capabilities the `-n` option is required. To run IDLE with the `-n` option follow the following instructions:

- Click on `Start`
- Click on `Programs`
- Click on `Python 2.3 (Enthought Edition)`
- Right click on `IDLE (Python GUI)` and click on `Create Shortcut` which creates `IDLE (Python GUI)(2)`
- Right click on `IDLE (Python GUI)(2)`
- Click on properties

- Add the `-n` option to `Target` being sure to leave a space after the last quote and before `-n`
- Drag and drop `IDLE (Python GUI)(2)` to the desktop to create an icon.

Step 8: Import a module and view the results

You can use the IDLE file browser to open the examples and run the module.

- Click on `File`
- Click on `Open`
- Open `C:\Path\To\Base\FiPy\directory\examples\diffusion\steadyState\mesh50x50\input.py`

This will open an editor, press the F5 key to run the example. A Matplotlib viewer should appear with rainbow stripes.

Another way to use FiPy examples is to import the example file and then run any functionality it may have. For example at the IDLE command line run the following code:

```
>>> import sys
>>> sys.path.append('C://Path//To//Base//FiPy//Directory')
>>> from examples.levelSet.electroChem.inputGold import runGold
>>> runGold(numberOfSteps = 100)
```

A MayaVi viewer should appear with a trench filling simulation.

2.8 Subversion tags

This section describes the use and setup of the FiPy Subversion repository. Users can download various tagged revisions of FiPy depending on their needs. In general, most users will not want to download the very latest state of FiPy, as these files are subject to active development and may not behave as desired. Most users will not be interested in particular version numbers either, but instead with the degree of code stability. Different "tracking tags" are used to indicate different stages of FiPy development. You will need to decide on your own risk tolerance when deciding which stage of development to track. A fresh copy of FiPy that is designated by a particular `<tag>` can be obtained with:

```
$ svn checkout http://matdl-osi.org/svn/fipy/<tag>
```

An existing SVN checkout of FiPy can be shifted to a different state of development by issuing the command:

```
$ svn switch http://matdl-osi.org/svn/fipy/<tag> .
```

in the base directory of the working copy. The main tags (`<tag>`) for FiPy, in decreasing order of stability, are:

`tags/STABLE` designates the most recent release in the repository that can be considered stable for daily use by the common user. This is a good tag to track if you don't want to run into bugs introduced with ongoing development but would like to take advantage of new features as soon as possible.

`tags/CURRENT` designates the most recent code on the trunk forming a coherent state of FiPy, in general this will mean a release, but can also mean a pre-release testing version. For instance, the release engineer might ask testers to test `tags/CURRENT` before he makes a release.

trunk designates the latest revision of any file present in the repository. FiPy is not guaranteed to pass its
    tests or to be in a consistent state when checked out under this tag.

tags/version-x_y designates a released version x.y. Any released version of FiPy will be designated with
    a fixed tag: The current version of FiPy is 1.1.

branches/branch-version-x_y designates a line of development based on a previously released version (i.e.,
    if current development work is being spent on version 0.2, but a bug is found and fixed in version 0.1,
    that fix will be tagged as version-0_1_1, and can be obtained from branch-version-0_1).

Any other tags will not generally be of interest to most users.

For those who are interested in learning more about Subversion, the canonical manual is the online Subversion
Red Bean book [15].

Chapter 3

Theoretical and Numerical Background

This chapter describes the numerical methods used to solve equations in the FiPy programming environment. FiPy uses the finite volume method (FVM) to solve coupled sets of partial differential equations (PDEs). For a good introduction to the FVM see Nick Croft's PhD thesis [19], Patanker [20] or Versteek and Malalasekera [21].

Essentially, the FVM consists of dividing the solution domain into discrete finite volumes over which the state variables are approximated with linear or higher order interpolations. The derivatives in each term of the equation are satisfied with simple approximate interpolations in a process known as discretization. The (FVM) is a popular discretization technique employed to solve coupled PDEs used in many application areas (*e.g.* Fluid Dynamics).

The FVM can be thought of as a subset of the Finite Element Method (FEM), just as the Finite Difference Method (FDM) is a subset of the FVM. A system of equations fully equivalent to the FVM can be obtained with the FEM using as weighting functions the characteristic functions of FV cells, i.e., functions equal to unity [22]. Analogously, the the discretization of equations with the FVM reduces to the FDM on Cartesian grids.

3.1 General Conservation Equation

The equations that model the evolution of physical, chemical and biological systems often have a remarkably universal form. Indeed, PDEs have proven necessary to model complex physical systems and processes that involve variations in both space and time. In general, given a variable of interest ϕ such as species concentration, pH, or temperature, there exists an evolution equation of the form

$$\frac{\partial \phi}{\partial t} = H(\phi, \lambda_i) \tag{3.1}$$

where H is a function of ϕ, other state variables λ_i, and higher order derivatives of all of these variables. Examples of such systems are wide ranging, but include problems that exhibit a combination of diffusing and reacting species, as well as such diverse problems as determination of the electric potential in heart tissue, of fluid flow, stress evolution, and even the Schrödinger equation.

A general conservation equation, solved using FiPy, can include any combination of the following terms,

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot (\vec{u}\phi)}_{\text{convection}} + \underbrace{\left[\nabla \cdot (\Gamma_i \nabla)\right]^n \phi}_{\text{diffusion}} + \underbrace{S_\phi}_{\text{source}} \tag{3.2}$$

where ρ, $\vec{u}$ and Γ_i represent coefficients in the transient, convection and diffusion terms, respectively. These coefficients can be arbitrary functions of any parameters or variables in the system. The variable ϕ represents

Figure 3.1: A mesh consists of cells, faces and vertices. For the purposes of FiPy, the divider between two cells is known as a face for all dimensions.

the unknown quantity in the equation. The diffusion term can represent any higher order diffusion-like term, where the order is given by the exponent n. For example, the diffusion term can represent conventional Fickean diffusion [$i.e.$, $\nabla \cdot (\Gamma \nabla \phi)$] when the exponent $n = 1$ or a Cahn-Hilliard term [$i.e.$, $\nabla \cdot (\Gamma_1 \nabla [\nabla \cdot \Gamma_2 \nabla \phi)]$]) [23, 24, 25]] when $n = 2$. Of course, higher order terms ($n > 2$) are also possible.

3.2 Finite Volume Method

To use the FVM, the solution domain must first be divided into non-overlapping polyhedral elements or cells. A solution domain divided in such a way is generally known as a mesh (as we will see, a Mesh is also a FiPy object). A mesh consists of vertices, faces and cells (see Figure 3.1). In the FVM the variables of interest are averaged over control volumes (CVs). The CVs are either defined by the cells or are centered on the vertices.

Cell Centered FVM (CC-FVM)

In the CC-FVM the CVs are formed by the mesh cells with the cell center "storing" the average variable value in the CV, (see Figure 3.2). The face fluxes are approximated using the variable values in the two adjacent cells surrounding the face. This low order approximation has the advantage of being efficient and requiring matrices of low band width (the band width is equal to the number of cell neighbors plus one) and thus low storage requirement. However, the mesh topology is restricted due to orthogonality and conjunctionality requirements. The value at a face is assumed to be the average value over the face. On an unstructured mesh the face center may not lie on the line joining the CV centers, which will lead to an error in the face interpolation. FiPy currently only uses the CC-FVM.

Vertex Centered FVM (VC-FVM)

In the VC-FVM, the CV is centered around the vertices and the cells are divided into sub-control volumes that make up the main CVs (see Figure 3.2). The vertices "store" the average variable values over the CVs. The CV faces are constructed within the cells rather than using the cell faces as in the CC-FVM. The face fluxes use all the vertex values from the cell where the face is located to calculate interpolations. For this reason, the VC-FVM is less efficient and requires more storage (a larger matrix band width) than the CC-FVM. However, the mesh topology does not have the same restrictions as the CC-FVM. FiPy does not have a VC-FVM capability.
Future releases of FiPy will have both the CC-FVM and VC-FVM capabilities.

(b)
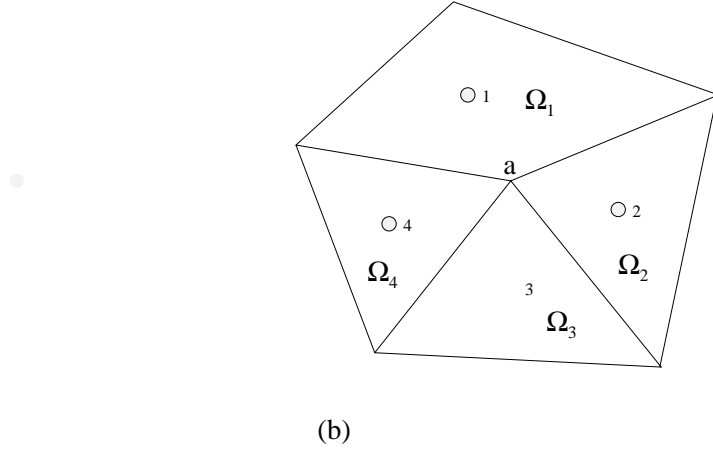
Figure 3.2: CV structure for an unstructured mesh, (a) Ω_a represents a vertex-based CV and (b) Ω_1, Ω_2, Ω_3 and Ω_4 represent cell centered CVs.

3.3 Discretization

The first step in the discretization of Equation (3.2) using the CC-FVM is to integrate over a CV and then make appropriate approximations for fluxes across the boundary of each CV. In this section, each term in Equation (3.2) will be examined separately.

Transient Term

For the transient term, the discretization of the integral $\int_V$ over the volume of a CV is given by

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV \simeq \frac{(\rho_P\phi_P - \rho_P^{\text{old}}\phi_P^{\text{old}})V_P}{\Delta t} \tag{3.3}$$

where ϕ_P represents the average value of ϕ in a CV centered on a point P and the superscript "old" represents the previous time-step value. The value V_P is the volume of the CV and Δt is the time step size.

Convection Term

The discretization for the convection term is given by

$$\int_V \nabla \cdot (\vec{u}\phi) \, dV \quad = \quad \int_S (\vec{n} \cdot \vec{u})\phi \, dS \tag{3.4}$$

$$\simeq \quad \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f \tag{3.5}$$

where we have used the divergence theorem to transform the integral over the CV volume $\int_V$ into an integral over the CV surface $\int_S$. The summation over the faces of a CV is denoted by $\sum_f$ and A_f is the area of each face. The vector $\vec{n}$ is the normal to the face pointing out of the CV into an adjacent CV centered on point A. When using a first order approximation, the value of ϕ_f must depend on the average value in adjacent cell ϕ_A and the average value in the cell of interest ϕ_P, such that

$$\phi_f = \alpha_f\phi_P + (1 - \alpha_f)\phi_A. \tag{3.6}$$

The weighting factor α_f is determined by the convection scheme, described later in this chapter.

Diffusion Term

The discretization for the diffusion term is given by

$$
\int_V \nabla \cdot (\Gamma \nabla \{\ldots\}) dV \quad = \quad \int_S \Gamma (\vec{n} \cdot \nabla \{\ldots\}) dS \tag{3.7}
$$

$$
\simeq \quad \sum_f \Gamma_f (\vec{n} \cdot \nabla \{\ldots\})_f A_f \tag{3.8}
$$

$\{\ldots\}$ indicates recursive application of the specified operation on ϕ, depending on the order of the diffusion term. The estimation for the flux, $(\vec{n} \cdot \nabla \{\ldots\})_f$, is obtained via

$$
(\vec{n} \cdot \nabla \{\ldots\})_f \simeq \frac{\{\ldots\}_A - \{\ldots\}_P}{d_{AP}} \tag{3.9}
$$

where the value of d_{AP} is the distance between neighboring cell centers. This estimate relies on the orthogonality of the mesh, and becomes increasingly inaccurate as the non-orthogonality increases. Correction terms have been derived to improve this error but are not currently included in FiPy [19].

Source Term

The discretization for the source term is given by,

$$
\int_V S_\phi \, dV \simeq S_\phi V_P. \tag{3.10}
$$

Including any negative dependence of S_ϕ on ϕ increases solution stability. The dependence can only be included in a linear manner so Equation (3.10) becomes

$$
V_P(S_0 + S_1 \phi_P), \tag{3.11}
$$

where S_0 is the source which is independent of ϕ and S_1 is the coeficient of the source which is linearly dependent on ϕ.

3.4   Linear Equations

The aim of the discretization is to reduce the continuous general equation to a set of discrete linear equations that can then be solved to obtain the value of the dependent variable at each CV center. This results in a sparse linear system that requires an efficient iterative scheme to solve. The iterative schemes available to FiPy are currently encapsulated in the spmatrix suite of solvers and include most common solvers such as the conjugate gradient method and LU decomposition. There are plans to include other solver suites that are compatible with Python.
Combining Equations (3.3), (3.5), (3.8) and (3.10), the complete discretization for equation (3.2) can now be written for each CV as

$$
\frac{\rho_P(\phi_P - \phi_P^{\mathrm{old}})V_P}{\Delta t} \quad = \quad \sum_f (\vec{n} \cdot \vec{u})_f A_f \left[\alpha_f \phi_P + (1 - \alpha_f) \phi_A \right]
$$

$$
+ \sum_f \Gamma_f A_f \frac{(\phi_A - \phi_P)}{d_{AP}} + V_P(S_0 + S_1 \phi_P). \tag{3.12}
$$

Equation (3.12) is now in the form of a set of linear combinations between each CV value and its neighboring values and can be written in the form

$$a_P \phi_P = \sum_f a_A \phi_A + b_P, \tag{3.13}$$

where

$$a_P = \frac{\rho_P V_P}{\Delta t} + \sum_f (a_A - F_f) - V_P S_1, \tag{3.14}$$

$$a_A = (1 - \alpha_f) F_f + D_f, \tag{3.15}$$

$$b_P = V_P S_0 + \frac{\rho_P V_P \phi_P^{\text{old}}}{\Delta t}. \tag{3.16}$$

The face coefficients, F_f and D_f, represent the convective strength and diffusive conductance respectively, and are given by

$$F_f = A_f (\vec{u} \cdot \vec{n})_f, \tag{3.17}$$

$$D_f = \frac{A_f \Gamma_f}{d_{AP}}. \tag{3.18}$$

3.5   Numerical Schemes

The coefficients of equation (3.13) must remain positive, since an increase in a neighboring value must result in an increase in ϕ_P to obtain physically realistic solutions. Thus, the inequalities $a_A > 0$ and $a_A - F_f > 0$ must be satisfied. The Péclet number $P_f \equiv -F_f/D_f$ is the ratio between convective strength and diffusive conductance. To achieve physically realistic solutions, the inequality

$$\frac{1}{1 - \alpha_f} > P_f > -\frac{1}{\alpha_f} \tag{3.19}$$

must be satisfied. The parameter α_f is defined by the chosen scheme, depending on Equation (3.19). The various differencing schemes are:

the central differencing scheme, where

$$\alpha_f = \frac{1}{2} \tag{3.20}$$

so that $|P_f| < 2$ satisfies Equation (3.19). Thus, the central differencing scheme is only numerically stable for a low values of P_f.

the upwind scheme, where

$$\alpha_f = \begin{cases} 1 & \text{if } P_f > 0, \\ 0 & \text{if } P_f < 0. \end{cases} \tag{3.21}$$

Equation (3.21) satisfies the inequality in Equation (3.19) for all values of P_f. However the solution over predicts the diffusive term leading to excessive numerical smearing ("false diffusion").

the exponential scheme, where

$$\alpha_f = \frac{(P_f - 1) \exp(P_f) + 1}{P_f (\exp(P_f) - 1)}. \tag{3.22}$$

This formulation can be derived from the exact solution, and thus, guarantees positive coefficients while not over-predicting the diffusive terms. However, the computation of exponentials is slow and therefore a faster scheme is generally used, especially in higher dimensions.

the hybrid scheme, where

$$
\alpha_f = \begin{cases}
\frac{P_f - 1}{P_f} & \text{if } P_f > 2, \\
\frac{1}{2} & \text{if } |P_f| < 2, \\
-\frac{1}{P_f} & \text{if } P_f < -2.
\end{cases}
\tag{3.23}
$$

The hybrid scheme is formulated by allowing $P_f \to \infty$, $P_f \to 0$ and $P_f \to -\infty$ in the exponential scheme. The hybrid scheme is an improvement on the upwind scheme, however, it deviates from the exponential scheme at $|P_f| = 2$.

the power law scheme, where

$$
\alpha_f = \begin{cases}
\frac{P_f - 1}{P_f} & \text{if } P_f > 10, \\
\frac{(P_f - 1) + (1 - P_f/10)^5}{P_f} & \text{if } 0 < P_f < 10, \\
\frac{(1 - P_f/10)^5 - 1}{P_f} & \text{if } -10 < P_f < 0, \\
-\frac{1}{P_f} & \text{if } P_f < -10.
\end{cases}
\tag{3.24}
$$

The power law scheme overcomes the inaccuracies of the hybrid scheme, while improving on the computational time for the exponential scheme.

All of the numerical schemes presented here are available in FiPy and can be selected by the user.

Chapter 4

Design and Implementation

The goal of FiPy is to provide a highly customizable, open source code for modeling problems involving coupled sets of PDEs. FiPy allows users to select and customize modules from within the framework. FiPy has been developed to address model problems in materials science such as poly-crystals, dendritic growth and electrochemical deposition. These applications all contain various combinations of PDEs with differing forms in conjunction with other unusual physics (over varying length scales) and unique solution procedures. The philosophy of FiPy is to enable customization while providing a library of efficient modules for common objects and data types.

4.1   Design

Numerical Approach

The solution algorithms given in the FiPy examples involve combining sets of PDEs while tracking an interface where the parameters of the problem change rapidly. The phase field method and the level set method are specialized techniques to handle the solution of PDEs in conjunction with a deforming interface. FiPy contains several examples of both methods.
FiPy uses the well-known Finite Volume Method (FVM) to reduce the model equations to a form tractable to linear solvers.

Object Oriented Structure

FiPy is programmed in an object-oriented manner. The benefit of object oriented programming mainly lies in encapsulation and inheritance. Encapsulation refers to the tight integration between certain pieces of data and methods that act on that data. Encapsulation allows parts of the code to be separated into clearly defined independent modules that can be re-applied or extended in new ways. Inheritance allows code to be reused, overridden, and new capabilities to be added without altering the original code. An object is treated by its users as an abstraction; the details of its implementation and behavior are internal.

Test Based Development

FiPy has been developed with a large number of test cases. These test cases are in two categories. The lower level tests operate on the core modules at the individual method level. The aim is that every method within the core installation has a test case. The high level test cases operate in conjunction with example solutions and serve to test global solution algorithms and the interaction of various modules.
With this two-tiered battery of tests, at any stage in code development, the test cases can be executed and errors can be identified. A comprehensive test base provides reassurance that any code breakages will be

clearly demonstrated with a broken test case. A test base also aids dissemination of the code by providing simple examples and knowledge of whether the code is working on a particular computer environment.

Open Source

In recent years, there has been a movement to release software under open source and associated unrestrictied licenses, especially within the scientific community. These licensing terms allow users to develop their own applications with complete access to the source code and then either contribute back to the main source repository or freely distribute their new adapted version.
As a product of the National Institute of Standards and Technology, the FiPy framework is placed in the public domain as a matter of U. S. Federal law. Furthermore, FiPy is built upon existing open source tools. Others are free to use FiPy as they see fit and we welcome contributions to make FiPy better.

High-Level Scripting Language

Programming languages can be broadly lumped into two categories: compiled languages and interpreted (or scripting) languages. Compiled languages are converted from a human-readable text source file to a machine-readable binary application file by a sequence of operations generally referred to as "compiling" and "linking". The binary application can then be run as many times as desired, but changes will provoke a new cycle of compiling and linking. Interpreted languages are converted from human-readable to machine-readable on the fly, each time the script is executed. Because the conversion happens every time[1], interpreted code is usually slower when running than compiled code. On the other hand, code development and debugging tends to be much easier and fluid when it's not necessary to wait for compile and link cycles after every change. Furthermore, because the conversion happens in real time, it is possible to have interactive sessions in a scripting language that are not generally possible in compiled languages.
Another distinction, somewhat orthogonal, but closely related, to that between compiled and interpreted languages, is between low-level languages and high-level languages. Low-level languages describe actions in simple terms that are closer to the way the computer actually functions. High-level languages describe actions in more complex and abstract terms that are closer to the way the programmer thinks about the problem at hand. This increased complexity in the meaning of an expression renders simpler code, because the details of the implementation are hidden away in the language internals or in an external library. For example, a low-level matrix multiplication written in C might be rendered as

```
if (Acols != Brows)
    error "these matrix shapes cannot be multiplied";

C = (float *) malloc(sizeof(float) * Bcols * Arows);

for (i = 0; i < Bcols; i++) {
    for (j = 0; j < Arows; j++) {
        C[i][j] = 0;
        for (k = 0; k < Acols; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Note that the dimensions of the arrays must be supplied externally, as C provides no intrinsic mechanism for determining the shape of an array. An equivalent high-level construction might be as simple as

---

[1] . . . neglecting such common optimizations as byte-code interpreters

```
C = A * B
```

All of the error checking, dimension measuring, and space allocation is handled automatically by low-level code that is intrinsic to the high-level matrix multiplication operator. The high-level code "knows" that matrices are involved, how to get their shapes, and to interpret '*' as a matrix multiplier instead of an arithmetic one. All of this allows the programmer to think about the operation of interest and not worry about introducing bugs in low-level code that is not unique to their application.

Although it needn't be true, for a variety of reasons, compiled languages tend to be low-level and interpreted languages tend to be high-level. Because low-level languages operate closer to the intrinsic "machine language" of the computer, they tend to be faster at running a given task than high-level languages, but programs written in them take longer to write and debug. Because running performance is a paramount concern, most scientific codes are written in low-level compiled languages like FORTRAN or C.

A rather common scenario in the development of scientific codes is that the first draft hard-codes all of the problem parameters. After a few (hundred) iterations of recompiling and relinking the application to explore changes to the parameters, code is added to read an input file containing a list of numbers. Eventually, the point is reached where it is impossible to remember which parameter comes in which order or what physical units are required, so code is added to, for example, interpret a line beginning with '#' as a comment. At this point, the scientist has begun developing a scripting language without even knowing it. Unfortunately for them, very few scientists have actually studied computer science or actually know anything about the design and implementation of script interpreters. Even if they have the expertise, the time spent developing such a language interpreter is time not spent actually doing research.

In contrast, a number of very powerful scripting languages, such as Tcl, Java, Python, Ruby, and even the venerable BASIC, have open source interpreters that can be embedded directly in an application, giving scientific codes immediate access to a high-level scripting language designed by someone who actually knew what they were doing.

We have chosen to go a step further and not just embed a full-fledged scripting language in the FiPy framework, but instead to design the framework from the ground up in a scripting language. While runtime performance is unquestionably important, many scientific codes are run relatively little, in proportion to the time spent developing them. If a code can be developed in a day instead of a month, it may not matter if it takes another day to run instead of an hour. Furthermore, there are a variety of mechanisms for diagnosing and optimizing those portions of a code that are actually time-critical, rather than attempting to optimize all of it by using a language that is more palatable to the computer than to the programmer. Thus FiPy, rather than taking the approach of writing the fast numerical code first and then dealing with the issue of user interaction, initially implements most modules in high-level scripting language and only translates to low-level compiled code those portions that prove inefficient. A discussion of efficiency issues can be found in Chapter 5.

Python Programming Language

Acknowledging that several scripting languages offer a number, if not all, of the features described above, we have selected Python for the implementation of FiPy. Python is:

- an interpreted language that combines remarkable power with very clear syntax,

- freely usable and distributable, even for commercial use,

- fully object oriented,

- distributed with powerful automated testing tools (doctest, unittest),

- actively used and extended by other scientists and mathemeticians (SciPy, Numeric, Scientific Python, PySparse).

- easily integrated with low-level languages such as C (weave, blitz, PyRex).

4.2   Implementation

The Python classes that make up FiPy are described in detail in the *FiPy Programmer's Reference*, but we give a brief overview here. FiPy is based around three fundamental Python classes: `Mesh`, `Variable`, and `Term`. Using the terminology of Chapter 3:

A `Mesh` object represents the domain of interest. FiPy contains many different specific mesh classes to describe different geometries.

A `Variable` object represents a quantity or field that can change during the problem evolution. A particular type of `Variable`, called a `CellVariable`, represents ϕ at the centers of the `Cell`s of the `Mesh`. A `CellVariable` describes the values of the field ϕ, but it is not concerned with their geometry; that role is taken by the `Mesh`.

An important property of `Variable` objects is that they can describe dependency relationships, such that:

```
>>> a = Variable(value = 3)
>>> b = a * 4
```

does not assign the value 12 to b, but rather it assigns a multiplication operator object to b, which depends on the `Variable` object a:

```
>>> b
(Variable(value = 3) * 4)
>>> a.setValue(5)
>>> b
(Variable(value = 5) * 4)
```

The numerical value of the `Variable` is not calculated until it is needed (a process known as "lazy evaluation"):

```
>>> print b
20
```

A `Term` object represents any of the terms in Equation (3.2) or any linear combination of such terms. Early in the development of FiPy, a distinction was made between `Equation` objects, which represented all of Equation (3.2), and `Term` objects, which represented the individual terms in that equation. The `Equation` object has since been eliminated as redundant. `Term` objects can be single entities such as an `ImplicitDiffusionTerm` or a linear combination of other `Term` objects that build up to form an expression such as Equation (3.2).

Beyond these three fundamental classes of `Mesh`, `Variable`, and `Term`, FiPy is composed of a number of related classes. The relationships between these classes are shown in Figure 4.1. A `Mesh` object is composed of `Cell` objects. Each `Cell` is defined by its bounding `Face` objects and each `Face` is defined by its bounding `Vertex` objects. A `Term` object encapsulates the contributions to the `SparseMatrix` that defines the solution of an equation. `BoundaryCondition` objects are used to describe the conditions on the boundaries of the `Mesh`, and each `Term` interprets the `BoundaryCondition` objects as necessary to modify the `SparseMatrix`. An equation constructed from `Term` objects can apply a unique `Solver` to invert its `SparseMatrix` in the most expedient and stable fashion. At any point during the solution, a `Viewer` can be invoked to display the values of the solved `Variable` objects.
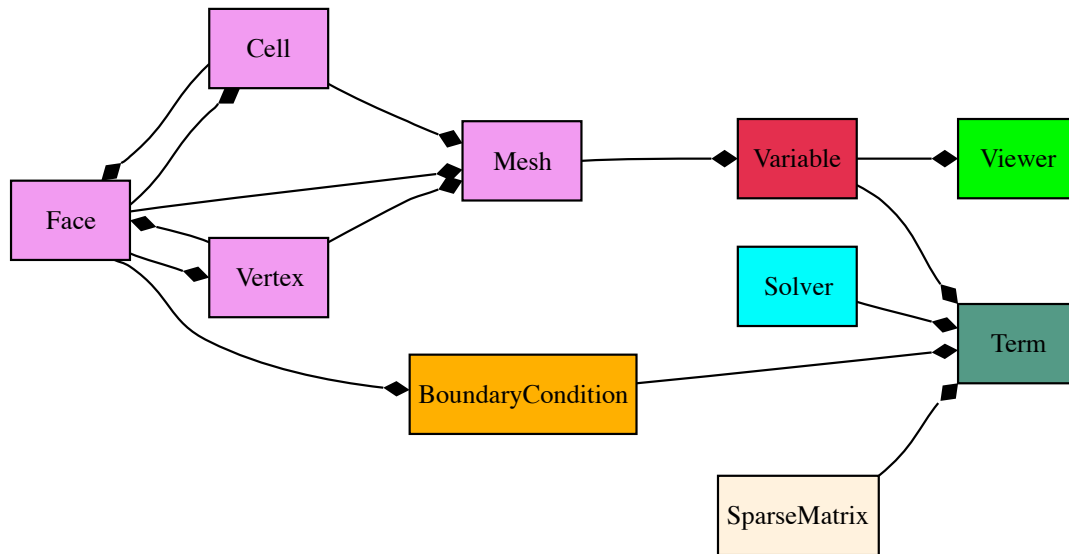
Figure 4.1: Primary object relationships in FiPy.

At this point, it will be useful to examine some of the example problems in Part II. More classes are introduced in the examples, along with illustrations of their instantiation and use.

Chapter 5

Efficiency

This section will present results and discussion of efficiency evaluations with FiPy. Programming in Python allows greater efficiency when designing and implementing new code, but it has some intrinsic inefficiencies during execution as compared with the C or FORTRAN programming languages. These inefficiencies can be minimized by translating sections of code that are used frequently into C.

FiPy has been tested against an in-house phase field code, written at NIST, to model grain growth and subsequent impingement. This problem can be executed by running:

```
$ examples/phase/impingement/mesh20x20/input.py \
> --numberOfElements=10000 --numberOfSteps=1000
```

from the base FiPy directory. The in-house code was written by Ryo Kobayashi and is used to generate the results presented in [26].

The raw CPU execution times for 10 time steps are presented in the following table. The run times are in seconds and the memory usage is in kilobytes. The Kobayashi code is given the heading of FORTRAN while FiPy is run with and without inlining. The memory usage is for FiPy simulations with the --inline. The --no-cache flag is on in all cases for the following table.

Elements	FiPy (s)	FiPy --inline (s)	FORTRAN (s)	FiPy memory (KiB)	FORTRAN memory (KiB)
100	0.77	0.30	0.0009	39316	772
400	0.87	0.37	0.0031	39664	828
1600	1.4	0.65	0.017	40656	1044
6400	3.7	2.0	0.19	46124	1880
25600	19	10	1.3	60840	5188
102400	79	43	4.6	145820	18436

The plain Python version of FiPy, which uses Numeric for all array operations, is around 17 times slower than the FORTRAN code. Using the --inline flag, this penalty is reduced to about 9 times slower.

It is hoped that in future releases of FiPy the process of C inlining for `Variable` objects will be automated. This may result in some efficiency gains, greater than we are seeing for this particular problem since all the `Variable` objects will be inlined. Recent analysis has shown that a `Variable` with multiple operations could be up to 6 times faster at calculating its value when inlined.

As presented in the above table, memory usage was also recorded for each FiPy simulation. From the table, once base memory usage is subtracted, each cell requires approximately 1.4 kilobytes of memory. The measurement of the maximum memory spike is hard with dynamic memory allocation, so these figures should only be used as a very rough guide. The FORTRAN memory usage is exact since memory is not allocated dynamically.

5.1   Efficiency comparison between `--no-cache` and `--cache` flags

This table shows results for efficiency tests when using the caching flags. Examples with more variables involved in complex expressions show the largest improvement in memory usage. The `--no-cache` option mainly prevents intermediate variables created due to binary operations from caching their values. This results in large memory gains while not effecting run times substantially. The table below is with `--inline` switched on and with 102400 elements for each case. The `--no-cache` flag is the default option.

Example	time per step `--no-cache` (s)	time per step `--cache` (s)	memory per cell `--no-cache` (KiB)	memory per cell `--cache` (KiB)
Impingement 9.6	4.3	4.1	1.4	2.3
Anisotropy 9.4	3.5	3.2	1.1	1.9
Cahn-Hilliard	3.0	2.5	1.1	1.4
Superfill 10.5	62	62	2.0	2.8

Chapter 6

Frequently Asked Questions

6.1 How do I represent an equation in FiPy?

As explained in Chapter 3, the canonical governing equation that can be solved by FiPy for the dependent `CellVariable` ϕ is

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{transient}} = \underbrace{\nabla \cdot (\vec{u}\phi)}_{\text{convection}} + \underbrace{[\nabla \cdot (\Gamma_i \nabla)]^n \phi}_{\text{diffusion}} + \underbrace{S_\phi}_{\text{source}} \tag{3.2}$$

A physical problem can involve many different coupled governing equations, one for each variable. Numerous specific examples are presented in Part II, but let us examine this general expression term-by-term:

How do I represent a transient term $\partial(\rho\phi)/\partial t$?

```
>>> TransientTerm(coeff = rho)
```

Note We have specified neither the variable ϕ nor the time step. Both are handled when we actually solve the equation.

How do I represent a convection term $\nabla \cdot (\vec{u}\phi)$?

```
>>> <Specific>ConvectionTerm(coeff = u,
...                          diffusionTerm = diffTerm)
```

where `<Specific>` can be any of `CentralDiff`, `Exponential`, `Hybrid`, `PowerLaw`, `Upwind`, `ExplicitUpwind`, or `VanLeer`. The differences between these convection schemes are described in Section 3.5. The velocity coefficient `u` must be a `FaceVectorVariable`, or a constant vector in the form of a Python list or tuple, *e.g.* `(1,2)` for a vector in 2D.

> **Note** As discussed in Section 3.5, the convection schemes need to calculate a Péclet number, and therefore need to know about any diffusion term used in the problem. It is hoped that this dependency can be automated in the future.

> **Warning** `VanLeerConvectionTerm` not mentioned and no discussion of explicit forms.

How do I represent a diffusion term $\nabla \cdot (\Gamma_1 \nabla \phi)$?

Either

```
>>> ImplicitDiffusionTerm(coeff = Gamma1)
```

or

```
>>> ExplicitDiffusionTerm(coeff = Gamma1)
```

`ExplicitDiffusionTerm` is provided only for illustrative purposes. `ImplicitDiffusionTerm` is almost always preferred. It is theoretically possible to create an explicit diffusion term with

```
>>> (Gamma1 * phi.getFaceGrad()).getDivergence()
```

Unfortunately, in this form, any boundary conditions on ϕ will not be accounted for.

How do I represent a term $\nabla^4 \phi$ or $\nabla \cdot (\Gamma_1 \nabla (\nabla \cdot (\Gamma_2 \nabla \phi)))$ such as for Cahn-Hilliard?

```
>>> ImplicitDiffusionTerm(coeff = (Gamma1, Gamma2))
```

The number of elements supplied for `coeff` determines the order of the term.

What if the term isn't one of those?

Any term that cannot be written in one of the previous forms is considered a source S_ϕ. An explicit source is written in Python essentially as it appears in mathematical form, *e.g.*, $3\kappa^2 + b\sin\theta$ would be written

```
>>> 3 * kappa**2 + b * sin(theta)
```

> **Note** Functions like `sin()` can be obtained from the `fipy.tools.numerix` module.

> **Warning** Generally, things will not work as expected if the equivalent function is used from the Numeric library.

If, however, the source depends on the variable that is being solved for, it can be advantageous to linearize the source and cast part of it as an implicit source term, *e.g.*, $3\kappa^2 + \phi\sin\theta$ might be written as

```
>>> 3 * kappa**2 + ImplicitSourceTerm(coeff = sin(theta))
```

> **Warning** There are subtleties in properly linearizing a source term which are examined in Example 9.1.

How do I represent a ... term that *doesn't* involve the dependent variable?

It is important to realize that, even though an expression may superficially resemble one of those shown above, if the dependent variable *for that PDE* does not appear in the appropriate place, then that term should be treated as a source.

How do I represent a diffusive source?

If the governing equation for ϕ is

$$\frac{\partial \phi}{\partial t} = \nabla \cdot (D_1 \nabla \phi) + \nabla \cdot (D_2 \nabla \xi)$$

then the first term is a `TransientTerm` and the second term is an `ImplicitDiffusionTerm`, but the third term is simply an explicit source, which is written in Python as

```
>>> (D2 * xi.getFaceGrad()).getDivergence()
```

Higher order diffusive sources can be obtained by simply nesting the calls to `getFaceGrad()` and `getDivergence()`.

> **Note** We use `getFaceGrad()`, rather than `getGrad()`, in order to obtain a second-order spatial discretization of the diffusion term in ξ, consistent with the matrix that is formed by `ImplicitDiffusionTerm` for ϕ.

How do I represent a convective source?

The convection of an independent field ξ as in

$$\frac{\partial \phi}{\partial t} = \nabla \cdot (\vec{u}\xi)$$

can be rendered as

```
>>> (u * xi.getArithmeticFaceValue()).getDivergence()
```

when $\vec{u}$ is a `VectorFaceVariable` (preferred) or as

```
>>> (u * xi).getDivergence()
```

if $\vec{u}$ is a `VectorCellVariable`.

How do I represent a transient source?

The time-rate-of change of an independent variable ξ, such as in

$$\frac{\partial(\rho_1\phi)}{\partial t} = \frac{\partial(\rho_2\xi)}{\partial t}$$

does not have an abstract form in FiPy and should be discretized directly, in the manner of Equation (3.3), as

```
>>> TransientTerm(coeff = rho1) == rho2 * (xi - xi.getOld()) / timeStep
```

This technique is used in Example 9.4.

What if my term involves the dependent variable, but not where FiPy puts it?

Frequently, viewing the term from a different perspective will allow it to be cast in one of the canonical forms. For example, the third term in

$$\frac{\partial\phi}{\partial t} = \nabla\cdot(D_1\nabla\phi) + \nabla\cdot(D_2\phi\nabla\xi)$$

might be considered as the diffusion of the independent variable ξ with a mobility $D_2\phi$ that is a function of the dependent variable ϕ. For FiPy's purposes, however, this term represents the convection of ϕ, with a velocity $D_2\nabla\xi$, due to the counter-diffusion of ξ, so

```
>>> diffTerm = ImplicitDiffusionTerm(coeff = D1)
>>> convTerm = <Specific>ConvectionTerm(coeff = D2 * xi.getFaceGrad(),
...                                     diffusionTerm = diffTerm)
>>> eq = TransientTerm() == diffTerm + convTerm
```

What if the coefficient of a term depends on the variable that I'm solving for?

A non-linear coefficient, such as the diffusion coefficient in $\nabla\cdot[\Gamma_1(\phi)\nabla\phi] = \nabla\cdot[\Gamma_0\phi(1-\phi)\nabla\phi]$ is not a problem for FiPy. Simply write it as it appears:

```
>>> diffTerm = ImplicitDiffusionTerm(coeff = Gamma0 * phi * (1 - phi))
```

Note Due to the nonlinearity of the coefficient, it will probably be necessary to "sweep" the solution to convergence as discussed in FAQ 6.5.

6.2   How do I export data?

The way to save your calculations depends on how you plan to make use of the data. If you want to save it for "restart" (so that you can continue or redirect a calculation from some intermediate stage), then you'll want to "pickle" the Python data with the `dump` module. This is illustrated in Examples 9.4, 9.5, 9.6, and 10.8.

On the other hand, pickled FiPy data is of little use to anything besides Python and FiPy. If you want to import your calculations into another piece of software, whether to make publication-quality graphs or movies, or to perform some analysis, or as input to another stage of a multiscale model, then you can save your data as an ASCII text file of tab-separated-values with a `TSVViewer`. This is illustrated in Example 7.3.

6.3 How do I save a plot image?

Some of the viewers have a button or other mechanism in the user interface for saving an image file. Also, you can supply an optional keyword `filename` when you tell the viewer to `plot()`, *e.g.*

```
>>> viewer.plot(filename="myimage.ext")
```

which will save a file named `myimage.ext` in your current working directory. The type of image is determined by the file extension ".ext". Different viewers have different capabilities: Pygist accepts ".eps" (Encapsulated PostScript) and ".cgm" (Computer Graphics Metafile). Matplotlib accepts ".eps", ".jpg" (Joint Photographic Experts Group), and ".png" (Portable Network Graphics). MayaVi only accepts ".png".

6.4 How do I make a movie?

FiPy has no facilities for making movies. You will need to save individual frames (see the previous question) and then stitch them together into a movie, using one of a variety of different free, shareware, or commercial software packages.

6.5 Iterations, timesteps, and sweeps? Oh, my!

Any non-linear solution of partial differential equations is an approximation. These approximations benefit from repetetive solution to achieve the best possible answer. In FiPy (and in many similar PDE solvers), there are three layers of repetition.

iterations This is the lowest layer of repetition, which you'll generally need to spend the least time thinking about. FiPy solves PDEs by discretizing them into a set of linear equations in matrix form, as explained in Sections 3.3 and 3.4. It is not always practical, or even possible, to exactly solve these matrix equations on a computer. FiPy thus employs "iterative solvers", which make successive approximations until the linear equations have been satisfactorily solved. FiPy chooses a default number of iterations and solution tolerance, which you will not generally need to change. If you do wish to change these defaults, you'll need to create a new `Solver` object with the desired number of iterations and solution tolerance, *e.g.*

```
>>> mySolver = LinearPCGSolver(iterations=1234, tolerance=5e-6)
    :
    :
>>> eq.solve(..., solver=mySolver, ...)
```

> **Note** The older `Solver` `steps=` keyword is now deprecated in favor of `iterations=` to make the role clearer.

Solver iterations are changed from their defaults in Examples 11.1, 12.1, and 13.

sweeps This middle layer of repetition is important when a PDE is non-linear (*e.g.*, a diffusivity that depends on concentration) or when multiple PDEs are coupled (*e.g.*, if solute diffusivity depends on temperature and thermal conductivity depends on concentration). Even if the `Solver` solves the *linear* approximation of the PDE to absolute perfection by performing an infinite number of iterations, the solution may still not be a very good representation of the actual *non-linear* PDE. If we resolve the same equation *at the same point in elapsed time*, but use the result of the previous solution instead of the previous timestep, then we can get a refined solution to the *non-linear* PDE in a process known as "sweeping."

> **Note** Despite references to the "previous timestep," sweeping is not limited to time-evolving problems. Nonlinear sets of quasi-static or steady-state PDEs can require sweeping, too.

We need to distinguish between the value of the variable at the last timestep and the value of the variable at the last sweep (the last cycle where we tried to solve the *current* timestep). This is done by first modifying the way the variable is created:

```
>>> myVar = CellVariable(..., hasOld=1)
```

and then by explicitly moving the current value of the variable into the "old" value only when we want to:

```
>>> myVar.updateOld()
```

Finally, we will need to repeatedly solve the equation until it gives a stable result. To clearly distinguish that a single cycle will not truly "solve" the equation, we invoke a different method "`sweep()`":

```
>>> for sweep in range(sweeps):
...     eq.sweep(var=myVar, ...)
```

Even better than sweeping a fixed number of cycles is to do it until the non-linear PDE has been solved satisfactorily:

```
>>> while residual > desiredResidual:
...     residual = eq.sweep(var=myVar, ...)
```

Sweeps are used to achieve better solutions in Examples 7.1, 9.1, 9.2, and 12.1.

timesteps This outermost layer of repetition is of most practical interest to the user. Understanding the time evolution of a problem is frequently the goal of studying a particular set of PDEs. Moreover, even when only an equilibrium or steady-state solution is desired, it may not be possible to simply solve that directly, due to non-linear coupling between equations or to boundary conditions or initial conditions. Some types of PDEs have fundamental limits to how large a timestep they can take before they become either unstable or inaccurate.

> **Note** Stability and accuracy are distinctly different. An unstable solution is often said to "blow up", with radically different values from point to point, often diverging to infinity. An inaccurate solution may look perfectly reasonable, but will disagree significantly from an analytical solution or from a numerical solution obtained by taking either smaller or larger timesteps.

For all of these reasons, you will frequently need to advance a problem in time and to choose an appropriate interval between solutions. This can be simple:

```
>>> timeStep = 1.234e-5
>>> for step in range(steps):
...     eq.solve(var=myVar, dt=timeStep, ...)
```

or more elaborate:

```
>>> timeStep = 1.234e-5
>>> elapsedTime = 0
>>> while elapsedTime < totalElapsedTime:
...     eq.solve(var=myVar, dt=timeStep, ...)
...     elapsedTime += timeStep
...     timeStep = SomeFunctionOfVariablesAndTime(myVar1, myVar2, elapedTime)
```

A majority of the examples in this manual illustrate time evolving behavior. Notably, boundary conditions are made a function of elapsed time in Example 7.1. The timestep is chosen based on the expected interfacial velocity in Example 9.1. The timestep is gradually increased as the kinetics slow down in Example 11.1.

Finally, we can (and often do) combine all three layers of repetition:

```
>>> myVar = CellVariable(..., hasOld=1)
    :
    :
>>> mySolver = LinearPCGSolver(iterations=1234, tolerance=5e-6)
    :
    :
>>> while elapsedTime < totalElapsedTime:
...     myVar.updateOld()
...     while residual > desiredResidual:
...         residual = eq.sweep(var=myVar, dt=timeStep, ...)
...     elapsedTime += timeStep
```

6.6 Why the distinction between `CellVariable` and `FaceVariable` coefficients?

FiPy solves field variables on the `Cell` centers. Transient and source terms describe the change in the value of a field at the `Cell` center, and so they take a `CellVariable` coefficient. Diffusion and convection terms involve fluxes *between* `Cell` centers, and are calculated on the `Face` between two `Cells`, and so they take a `FaceVariable` or `VectorFaceVariable` coefficient.

Note If you supply a `CellVariable` var when a `FaceVariable` is expected, FiPy will automatically substitute `var.getArithmeticFaceValue()`. This can have undesirable consequences, however. For one thing, the arithmetic face average of a non-linear function is not the same as the same non-linear function of the average argument, *e.g.*, for $f(x) = x^2$,

$$f(\frac{1+2}{2}) = \frac{9}{4} \neq \frac{f(1) + f(2)}{2} = \frac{5}{2}$$

This distinction is not generally important for smoothly varying functions, but can dramatically affect the solution when sharp changes are present. Also, for many problems, such as a conserved concentration field that cannot be allowed to drop below zero, a harmonic average is more appropriate than an arithmetic average. If you experience problems (unstable or wrong results, or excessively small timesteps), you may need to explicitly supply the desired `FaceVariable` rather than letting FiPy assume one.

6.7 What does this error message mean?

- "`ValueError:  frames are not aligned`"

  This error most likely means that you have provided a `CellVariable` when FiPy was expecting a `FaceVariable` (or vice versa).

- "`MA.MA.MAError:  Cannot automatically convert masked array to Numeric because data is masked in one or more locations.`"

This not-so-helpful error message could mean a number of things, but the most likely explanation is that the solution has become unstable and is diverging to $\pm\infty$. This can be caused by taking too large a timestep or by using explicit terms instead of implicit ones.

- "`repairing catalog by removing key`"

  This message (not really an error, but may cause test failures) can result when using the SciPy weave package via the `--inline` flag. It is due to a bug in SciPy that has been patched in their source repository: `http://www.scipy.org/mailinglists/mailman?fn=scipy-dev/2005-June/003010.html`.

- "`numerix Numeric 23.6`"

  This is neither an error nor a warning. It's just sloppy message left in SciPy: `http://thread.gmane.org/gmane.comp.python.scientific.user/4349`.

6.8   What if my question isn't answered here?

Please post your question to the mailing list [6] or file a Tracker request [7].

Part II

Examples

Note

Any given "Module example.something.input" can be found in the file "examples/something/input.py".

These examples can be used in at least four ways:

- Each example can be invoked individually to demonstrate an application of FiPy:

  ```
  $ examples/something/input.py
  ```

- Each example can be invoked such that when it has finished running, you will be left in an interactive Python shell:

  ```
  $ python -i examples/something/input.py
  ```

  At this point, you can enter Python commands to manipulate the model or to make queries about the example's variable values. For instance, the interactive Python sessions in the example documentation can be typed in directly to see that the expected results are obtained.

- Alternatively, these interactive Python sessions, known as doctest blocks, can be invoked as automatic tests:

  ```
  $ python setup.py test --examples
  ```

  In this way, the documentation and the code are always certain to be consistent.

- Finally, and most importantly, the examples can be used as a templates to design your own problem scripts.

  Note

  The examples shown in this manual have been written with particular emphasis on serving as both documentation and as comprehensive tests of the FiPy framework. As explained at the end of examples/diffusion/steadyState/mesh1D.py, your own scripts can be much more succint, if you wish, and include only the text that follows the ">>>" and "..." prompts shown in these examples. To obtain a copy of an example, containing just the script instructions, type:

  ```
  $ python setup.py copy_script --From x.py --To y.py
  ```

In addition to those presented in this manual, there are dozens of other files in the examples/ directory (all with "input" in their title), that demonstrate other uses of FiPy. If these examples do not help you construct your own problem scripts, please contact us.

Example Contents

Chapter 7

Diffusion Examples

7.1 Module `examples.diffusion.mesh1D`

To run this example from the base FiPy directory, type:

```
$ examples/diffusion/mesh1D.py
```

at the command line. Different stages of the example should be displayed, along with prompting messages in the terminal.

This example takes the user through assembling a simple problem with FiPy. It describes different approaches to a 1D diffusion problem with constant diffusivity and fixed value boundary conditions such that,

$$\frac{\partial \phi}{\partial t} = D\nabla^2 \phi. \tag{7.1}$$

The first step is to define a one dimensional domain with 50 solution points. The `Grid1D` object represents a linear structured grid. The parameter `dx` refers to the grid spacing (set to unity here).

```
>>> nx = 50
>>> dx = 1.
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(nx = nx, dx = dx)
```

FiPy solves all equations at the centers of the cells of the mesh. We thus need a `CellVariable` object to hold the values of the solution, with the initial condition $\phi = 0$ at $t = 0$,

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phi = CellVariable(name="solution variable",
...                    mesh=mesh,
...                    value=0)
```

We'll let

```
>>> D = 1.
```

for now.

The set of boundary conditions are given to the equation as a Python `tuple` or `list` (the distinction is not generally important to FiPy). The boundary conditions

$$\phi = \begin{cases} 0 & \text{at } x = 1, \\ 1 & \text{at } x = 0. \end{cases}$$

are formed with a value

```
>>> valueLeft = 1
>>> valueRight = 0
```

and a set of faces over which they apply.

> **Note**
>
> Only faces around the exterior of the mesh can be used for boundary conditions.

For example, here the exterior faces on the left of the domain are extracted by `mesh.getFacesLeft()`. A `FixedValue` boundary condition is created with these faces and a value (`valueLeft`).

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> BCs = (FixedValue(faces=mesh.getFacesRight(), value=valueRight),
...        FixedValue(faces=mesh.getFacesLeft(), value=valueLeft))
```

> **Note**
>
> If no boundary conditions are specified on exterior faces, the default boundary condition is
> `FixedFlux(faces=someFaces, value=0.)`, equivalent to $\vec{n} \cdot \nabla\phi|_{\mathsf{someFaces}} = 0$.

If you have ever tried to numerically solve Eq. (7.1), you most likely attempted "explicit finite differencing" with code something like:

```
for step in range(steps):
    for j in range(cells):
        phi_new[j] = phi_old[j] \
            + (D * dt / dx**2) * (phi_old[j+1] - 2 * phi_old[j] + phi_old[j-1])
    time += dt
```

plus additional code for the boundary conditions. In FiPy, you would write

```
>>> from fipy.terms.explicitDiffusionTerm import ExplicitDiffusionTerm
>>> from fipy.terms.transientTerm import TransientTerm
>>> eqX = TransientTerm() == ExplicitDiffusionTerm(coeff=D)
```

The largest stable timestep that can be taken for this explicit 1D diffusion problem is $\Delta t \leq \Delta x^2/(2D)$. We limit our steps to 90% of that value for good measure

```
>>> timeStepDuration = 0.9 * dx**2 / (2 * D)
>>> steps = 100
```

In a semi-infinite domain, the analytical solution for this transient diffusion problem is given by $\phi = 1 - \mathrm{erf}(x/2\sqrt{Dt})$. If the SciPy library is available, the result is tested against the expected profile:

```
>>> x = mesh.getCellCenters()[...,0]
>>> t = timeStepDuration * steps
>>> from fipy.tools.numerix import sqrt

>>> phiAnalytical = CellVariable(name="analytical value",
...                              mesh=mesh)

>>> try:
...     from scipy.special import erf
...     phiAnalytical.setValue(1 - erf(x / (2 * sqrt(D * t))))
... except ImportError:
...     print "The SciPy library is not available to test the solution to \
... the transient diffusion equation"
```

If we're running interactively, we'll want to view the result, but not if this example is being run automatically as a test. We accomplish this by having Python check if this script is the "`__main__`" script, which will only be true if we explicitly launched it and not if it has been imported by another script such as the automatic tester. The function `fipy.viewers.make()` returns a suitable viewer depending on available viewers and the dimension of the mesh.

```
>>> if __name__ == '__main__':
...     from fipy import viewers
...     viewer = viewers.make(vars=(phi, phiAnalytical),
...                           limits={'datamin': 0., 'datamax': 1.})
...     viewer.plot()
```

We then solve the equation by repeatedly looping in time:

```
>>> for step in range(steps):
...     eqX.solve(var=phi,
...               boundaryConditions=BCs,
...               dt=timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()

>>> print phi.allclose(phiAnalytical, atol = 7e-4)
1

>>> if __name__ == '__main__':
...     raw_input("Explicit transient diffusion. Press <return> to proceed...")
```
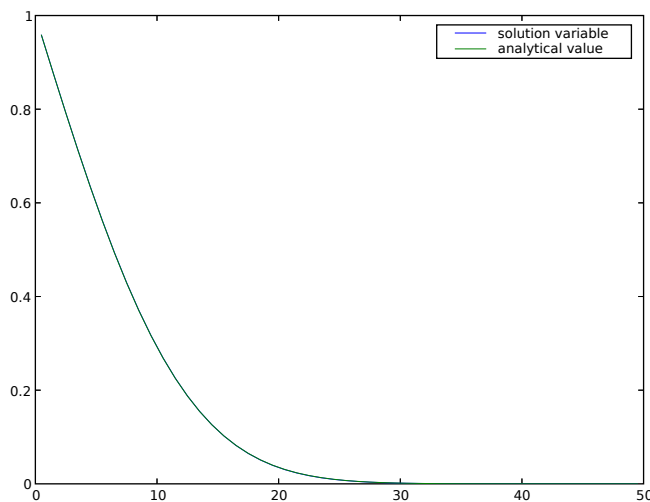

Although explicit finite differences are easy to program, we have just seen that this 1D transient diffusion problem is limited to taking rather small time steps. If, instead, we represent Eq. (7.1) as:

```
phi_new[j] = phi_old[j] \
   + (D * dt / dx**2) * (phi_new[j+1] - 2 * phi_new[j] + phi_new[j-1])
```

it is possible to take much larger time steps. Because `phi_new` appears on both the left and right sides of the equation, this form is called "implicit". In general, the "implicit" representation is much more difficult to program than the "explicit" form that we just used, but in FiPy, all that is needed is to write

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> eqI = TransientTerm() == ImplicitDiffusionTerm(coeff=D)
```

reset the problem

```
>>> phi.setValue(valueRight)
```

and rerun with much larger time steps

```
>>> timeStepDuration *= 10
>>> steps /= 10
>>> for step in range(steps):
...     eqI.solve(var=phi,
...               boundaryConditions=BCs,
...               dt=timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()

>>> print phi.allclose(phiAnalytical, atol = 2e-2)
1

>>> if __name__ == '__main__':
...     raw_input("Implicit transient diffusion. Press <return> to proceed...")
```
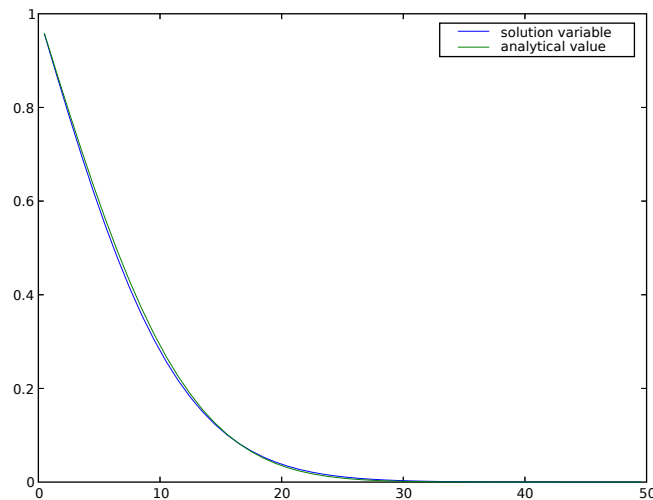

Note that although much larger *stable* timesteps can be taken with this implicit version (there is, in fact, no limit to how large an implicit timestep you can take for this particular problem), the solution is less *accurate*. One way to achieve a compromise between *stability* and *accuracy* is with the Crank-Nicholson scheme, represented by:

```
phi_new[j] = phi_old[j] + (D * dt / (2 * dx**2)) * \
                ((phi_new[j+1] - 2 * phi_new[j] + phi_new[j-1])
                 + (phi_old[j+1] - 2 * phi_old[j] + phi_old[j-1]))
```

which is essentially an average of the explicit and implicit schemes from above. This can be rendered in FiPy as easily as

```
>>> eqCN = eqX + eqI
```

We again reset the problem

```
>>> phi.setValue(valueRight)
```

and apply the Crank-Nicholson scheme until the end, when we apply one step of the fully implicit scheme to drive down the error (see, *e.g.*, [27, §19.2]).

```
>>> for step in range(steps - 1):
...     eqCN.solve(var=phi,
...                boundaryConditions=BCs,
...                dt=timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()
>>> eqI.solve(var=phi,
...           boundaryConditions=BCs,
...           dt=timeStepDuration)
>>> if __name__ == '__main__':
...     viewer.plot()

>>> print phi.allclose(phiAnalytical, atol = 3e-3)
1

>>> if __name__ == '__main__':
...     raw_input("Crank-Nicholson transient diffusion. Press <return> to proceed...")
```

As mentioned above, there is no stable limit to how large a time step can be taken for the implicit diffusion problem. In fact, if the time evolution of the problem is not interesting, it is possible to eliminate the time step altogether by omitting the `TransientTerm`. The steady-state diffusion equation

$$D\nabla^2\phi = 0$$

is represented in FiPy by

```
>>> ImplicitDiffusionTerm(coeff=D).solve(var=phi,
...                                      boundaryConditions=BCs)
...

>>> if __name__ == '__main__':
...     viewer.plot()
```

The analytical solution to the steady-state problem is no longer an error function, but simply a straight line, which we can confirm to a tolerance of 10^{-10}.

```
>>> L = nx * dx
>>> print phi.allclose(valueLeft + (valueRight - valueLeft) * x / L,
...                     rtol = 1e-10, atol = 1e-10)
1

>>> if __name__ == '__main__':
...     raw_input("Implicit steady-state diffusion. Press <return> to proceed...")
```
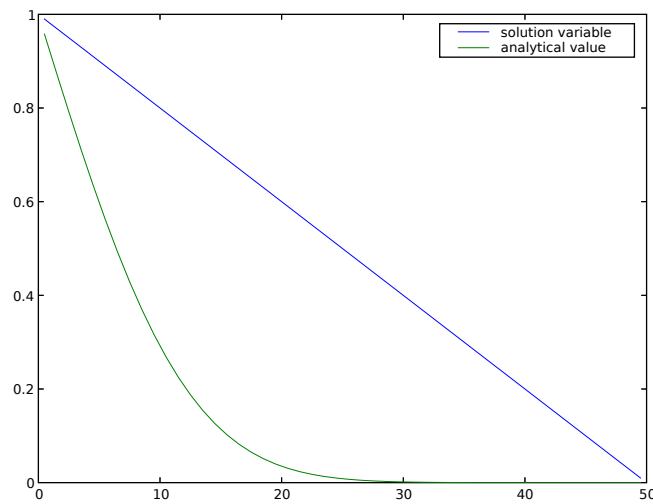

Often, boundary conditions may be functions of another variable in the system or of time. For example, to have

$$\phi = \begin{cases} (1 + \sin t)/2 & \text{on } x = 0 \\ 0 & \text{on } x = L \end{cases}$$

we will need to declare time t as a `Variable`

```
>>> from fipy.variables.variable import Variable
>>> time = Variable()
```

and then declare our boundary condition as a function of this `Variable`

```
>>> from fipy.tools.numerix import sin
>>> BCs = (FixedValue(faces=mesh.getFacesLeft(), value=0.5 * (1 + sin(time))),
...        FixedValue(faces=mesh.getFacesRight(), value=0.))
```

When we update `time` at each timestep, the left-hand boundary condition will automatically update,

```
>>> dt = .1
>>> while time() < 15:
...     time.setValue(time() + dt)
...     eqI.solve(var=phi, dt=dt, boundaryConditions=BCs)
...     if __name__ == '__main__':
...         viewer.plot()
```

```
>>> if __name__ == '__main__':
...     raw_input("Time-dependent boundary condition. Press <return> to proceed...")
```
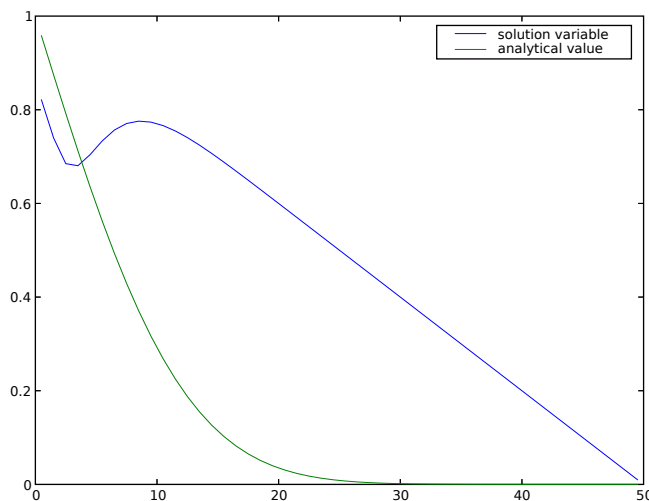

Many interesting problems do not have simple, uniform diffusivities. We consider a steady-state diffusion problem

$$\nabla \cdot (D\nabla\phi) = 0,$$

with a spatially varying diffusion coefficient

$$D = \begin{cases} 1 & \text{for } 0 < x < L/4, \\ 0.1 & \text{for } L/4 \leq x < 3L/4, \\ 1 & \text{for } 3L/4 \leq x < L, \end{cases}$$

and with boundary conditions $\phi = 0$ at $x = 0$ and $D\frac{\partial\phi}{\partial x} = 1$ at $x = L$, where L is the length of the solution domain. Exact numerical answers to this problem are found when the mesh has cell centers that lie at $L/4$ and $3L/4$, or when the number of cells in the mesh N_i satisfies $N_i = 4i + 2$, where i is an integer. The mesh we've been using thus far is satisfactory, with $Ni = 50$ and $i = 12$. Because FiPy considers diffusion to be a flux from one `Cell` to the next, through the intervening `Face`, we must define the non-uniform diffusion coefficient on the mesh faces

```
>>> from fipy.variables.faceVariable import FaceVariable
>>> D = FaceVariable(mesh=mesh, value=1.0)
>>> x = mesh.getFaceCenters()[...,0]
>>> D.setValue(0.1, where=(L / 4. <= x) & (x < 3. * L / 4.))
```

The boundary conditions are a fixed value of

```
>>> valueLeft = 0.
```

to the left and a fixed flux of

```
>>> fluxRight = 1.
```

to the right:

```
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> BCs = (FixedValue(faces=mesh.getFacesLeft(), value=valueLeft),
...            FixedFlux(faces=mesh.getFacesRight(), value=fluxRight))
```

We re-initialize the solution variable

```
>>> phi.setValue(0)
```

and obtain the steady-state solution with one implicit solution step

```
>>> ImplicitDiffusionTerm(coeff = D).solve(var=phi,
...                                           boundaryConditions = BCs)
```

The analytical solution is simply

$$\phi = \begin{cases} x & \text{for } 0 < x < L/4, \\ 10x - 9L/4 & \text{for } L/4 \le x < 3L/4, \\ x + 18L/4 & \text{for } 3L/4 \le x < L, \end{cases}$$

or

```
>>> x = mesh.getCellCenters()[...,0]
>>> from fipy.tools.numerix import where
>>> phiAnalytical.setValue(x)
>>> phiAnalytical.setValue(10 * x - 9. * L / 4. ,
...                        where=(L / 4. <= x) & (x < 3. * L / 4.))
>>> phiAnalytical.setValue(x + 18. * L / 4. ,
...                        where=3. * L / 4. <= x)
>>> print phi.allclose(phiAnalytical, atol = 1e-8, rtol = 1e-8)
1
```

And finally, we can plot the result

```
>>> if __name__ == '__main__':
...     viewers.make(vars=(phi, phiAnalytical)).plot()
...     raw_input("Non-uniform steady-state diffusion. Press <return> to proceed...")
```
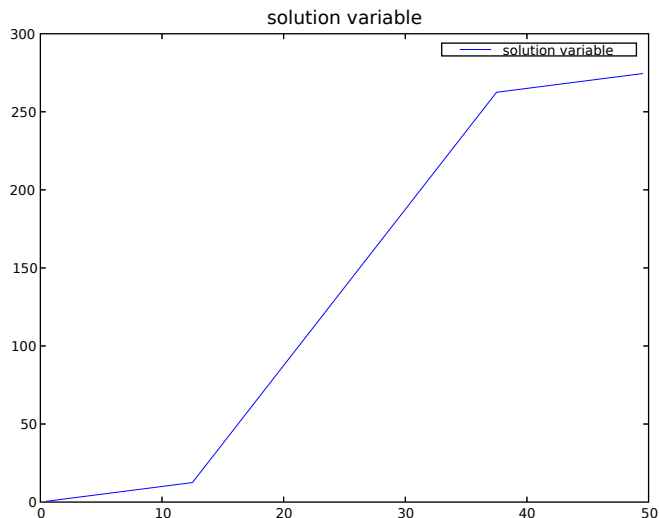
Often, the diffusivity is not only non-uniform, but also depends on the value of the variable, such that

$$\frac{\partial \phi}{\partial t} = \nabla \cdot [D(\phi)\nabla\phi]. \tag{7.2}$$

With such a non-linearity, it is generally necessary to "sweep" the solution to convergence. This means that each time step should be calculated over and over, using the result of the previous sweep to update the coefficients of the equation, without advancing in time. In FiPy, this is accomplished by creating a solution variable that explicitly retains its "old" value by specifying `hasOld` when you create it. The variable does not move forward in time until it is explicity told to `updateOld()`. In order to compare the effects of different numbers of sweeps, let us create a list of variables: `phi[0]` will be the variable that is actually being solved and `phi[1]` through `phi[4]` will display the result of taking the corresponding number of sweeps (`phi[1]` being equivalent to not sweeping at all).

```
>>> valueLeft = 1
>>> valueRight = 0
>>> phi = [
...     CellVariable(name="solution variable",
...                  mesh=mesh,
...                  value=valueRight,
...                  hasOld=1),
...     CellVariable(name="1 sweep",
...                  mesh=mesh),
...     CellVariable(name="2 sweeps",
...                  mesh=mesh),
...     CellVariable(name="3 sweeps",
...                  mesh=mesh),
...     CellVariable(name="4 sweeps",
...                  mesh=mesh)
... ]
```

If, for example,

$$D = D_0(1 - \phi)$$

we would simply write Eq. (7.2) as

```
>>> D0 = 1
>>> eq = TransientTerm() == ImplicitDiffusionTerm(coeff=D0 * (1 - phi[0]))
```

Note

Because of the non-linearity, the Crank-Nicholson scheme does not work for this problem.

We apply the same boundary conditions that we used for the uniform diffusivity cases

```
>>> BCs = (FixedValue(faces=mesh.getFacesRight(), value=valueRight),
...             FixedValue(faces=mesh.getFacesLeft(), value=valueLeft))
```

Although this problem does not have an exact transient solution, it can be solved in steady-state, with

$$\phi(x) = 1 - \sqrt{\frac{x}{L}}$$

```
>>> x = mesh.getCellCenters()[...,0]
>>> phiAnalytical.setValue(1. - sqrt(x/L))
```

We create a viewer to compare the different numbers of sweeps with the analytical solution from before.

```
>>> if __name__ == '__main__':
...     viewer = viewers.make(vars=phi + [phiAnalytical],
...                           limits={'datamin': 0., 'datamax': 1.})
...     viewer.plot()
```

As described above, an inner "sweep" loop is generally required for the solution of non-linear or multiple equation sets. Often a conditional is required to exit this "sweep" loop given some convergence criteria. Instead of using the `solve()` method equation, when sweeping, it is often useful to call `sweep()` instead. The `sweep()` method behaves the same way as `solve()`, but returns the residual that can then be used as part of the exit condition.
We now repeatedly run the problem with increasing numbers of sweeps.

```
>>> for sweeps in range(1,5):
...     phi[0].setValue(valueRight)
...     for step in range(steps):
...         # only move forward in time once per time step
...         phi[0].updateOld()
...
...         # but "sweep" many times per time step
...         for sweep in range(sweeps):
...             res = eq.sweep(var=phi[0],
...                            boundaryConditions=BCs,
...                            dt=timeStepDuration)
...         if __name__ == '__main__':
...             viewer.plot()
...
...     # copy the final result into the appropriate display variable
```

```
...          phi[sweeps].setValue(phi[0])
...          if __name__ == '__main__':
...              viewer.plot()
...              raw_input("Implicit variable diffusity. %d sweep(s). \
... Residual = %f. Press <return> to proceed..." % (sweeps, max(abs(res))))
```

As can be seen, sweeping does not dramatically change the result, but the "residual" of the equation (a measure of how accurately it has been solved) drops about an order of magnitude with each additional sweep.

> **Attention!**
>
> Choosing an optimal balance between the number of time steps, the number of sweeps, the number of solver iterations, and the solver tolerance is more art than science and will require some experimentation on your part for each new problem.

Finally, we can increase the number of steps to approach equilibrium, or we can just solve for it directly

```
>>> eq = ImplicitDiffusionTerm(coeff=D0 * (1 - phi[0]))

>>> phi[0].setValue(valueRight)
>>> res = 1e+10
>>> while res > 1e-6:
...     res = eq.sweep(var=phi[0],
...                    boundaryConditions=BCs,
...                    dt=timeStepDuration)

>>> print phi[0].allclose(phiAnalytical, atol = 1e-1)
1

>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("Implicit variable diffusity - steady-state. \
... Press <return> to proceed...")
```
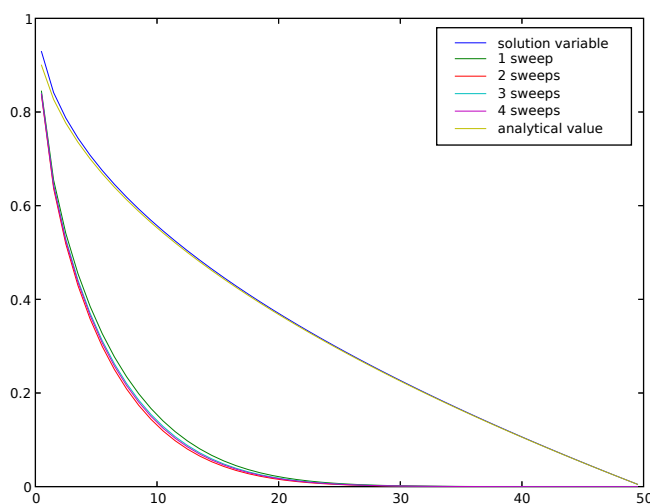
If this example had been written primarily as a script, instead of as documentation, we would delete every line that does not begin with either ">>>" or "...", and then delete those prefixes from the remaining lines, leaving:

```python
#!/usr/bin/env python

## This script was derived from
## 'examples/diffusion/mesh1D.py'

nx = 50
dx = 1.
from fipy.meshes.grid1D import Grid1D
mesh = Grid1D(nx = nx, dx = dx)
from fipy.variables.cellVariable import CellVariable
phi = CellVariable(name="solution variable",
                   mesh=mesh,
                   value=0)


    .
    .
    .


eq = ImplicitDiffusionTerm(coeff=D0 * (1 - phi[0]))
phi[0].setValue(valueRight)
res = 1e+10
while res > 1e-6:
    res = eq.sweep(var=phi[0],
                   boundaryConditions=BCs,
                   dt=timeStepDuration)

print phi[0].allclose(phiAnalytical, atol = 1e-1)
# Expect:
# 1
#
if __name__ == '__main__':
    viewer.plot()
    raw_input("Implicit variable diffusity - steady-state. \
Press <return> to proceed...")
```

Your own scripts will tend to look like this, although you can always write them as doctest scripts if you choose. You can obtain a plain script like this from some `.../example.py` by typing:

```
$ python setup.py copy_script --From .../example.py --To myExample.py
```

at the command line.

Most of the FiPy examples will be a mixture of plain scripts and doctest documentation/tests.

7.2   Module `examples.diffusion.mesh20x20`

This example again solves a 1D diffusion problem as in `examples/diffusion/mesh1D.py`, but now on a two-dimensional mesh:

```
>>> nx = 20
>>> ny = nx
>>> dx = 1.
>>> dy = dx
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

We create a `CellVariable` and initialize it to zero:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phi = CellVariable(name = "solution variable",
...                    mesh = mesh,
...                    value = 0)
```

and then create a diffusion equation. This is solved by default with an iterative conjugate gradient solver.

```
>>> D = 1.
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> eq = TransientTerm() == ImplicitDiffusionTerm(coeff=D)
```

We apply Dirichlet boundary conditions

```
>>> valueLeft = 1
>>> valueRight = 0
```

to the left and right. Neumann boundary conditions are automatically applied to the top and bottom.

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> BCs = (FixedValue(faces=mesh.getFacesLeft(),
...                   value=valueLeft),
...        FixedValue(faces=mesh.getFacesRight(),
...                   value=valueRight))
```

We create a viewer to see the results

```
>>> if __name__ == '__main__':
...     from fipy import viewers
...     viewer = viewers.make(vars=phi,
...                           limits={'datamin': 0., 'datamax': 1.})
...     viewer.plot()
```

and solve the equation by repeatedly looping in time:

```
>>> timeStepDuration = 10 * 0.9 * dx**2 / (2 * D)
>>> steps = 10
>>> for step in range(steps):
...     eq.solve(var=phi,
...              boundaryConditions=BCs,
...              dt=timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()
```
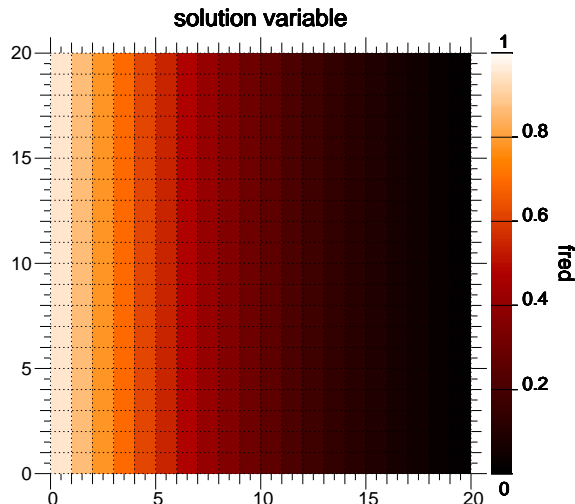
We can again test against the analytical solution $\phi = 1 - \text{erf}(x/2\sqrt{Dt})$.

```
>>> x = mesh.getCellCenters()[...,0]
>>> t = timeStepDuration * steps
>>> from fipy.tools.numerix import sqrt
```

```
>>> phiAnalytical = CellVariable(name="analytical value",
...                              mesh=mesh)
```

```
>>> try:
...     from scipy.special import erf
...     phiAnalytical.setValue(1 - erf(x / (2 * sqrt(D * t))))
... except ImportError:
...     print "The SciPy library is not available to test the solution to \
... the transient diffusion equation"
```

```
>>> print phi.allclose(phiAnalytical, atol = 4e-2)
1
```

```
>>> if __name__ == '__main__':
...     raw_input("Implicit transient diffusion. Press <return> to proceed...")
```

We can also solve the steady-state problem directly

```
>>> ImplicitDiffusionTerm().solve(var=phi,
...                               boundaryConditions = BCs)
>>> if __name__ == '__main__':
...     viewer.plot()
```
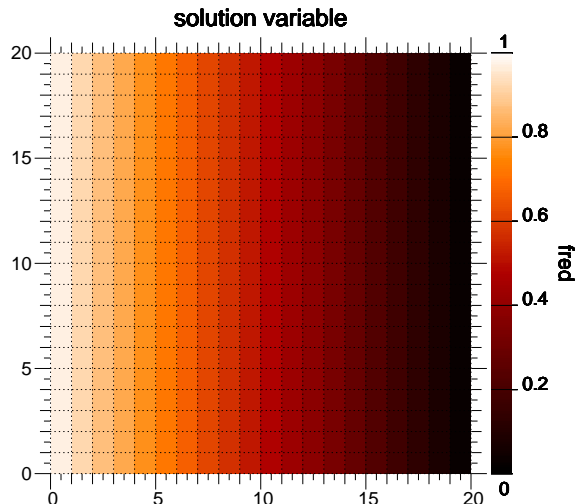
and test the result against the expected linear composition profile:

```
>>> L = nx * dx
>>> x = mesh.getCellCenters()[...,0]
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / L
>>> print phi.allclose(analyticalArray, rtol = 2e-10, atol = 2e-10)
1

>>> if __name__ == '__main__':
...     raw_input("Implicit steady-state diffusion. Press <return> to proceed...")
```

7.3 Module `examples.diffusion.circle`

This example demonstrates how to solve a simple diffusion problem on a non-standard mesh with varying boundary conditions. The gmsh package is used to create the mesh. Firstly, define some parameters for the creation of the mesh,

```
>>> cellSize = 0.05
>>> radius = 1.
```

The `cellSize` is the preferred edge length of each mesh element and the `radius` is the radius of the circular mesh domain. In the following code section a file is created with the geometry that describes the mesh. For details of how to write such geometry files for gmsh, see the gmsh manual.

```
>>> lines = [ 'cellSize = ' + str(cellSize) + ';\n',
...           'radius = ' + str(radius) + ';\n',
...           'Point(1) = {0, 0, 0, cellSize};\n',
...           'Point(2) = {-radius, 0, 0, cellSize};\n',
...           'Point(3) = {0, radius, 0, cellSize};\n',
...           'Point(4) = {radius, 0, 0, cellSize};\n',
...           'Point(5) = {0, -radius, 0, cellSize};\n',
...           'Circle(6) = {2, 1, 3};\n',
...           'Circle(7) = {3, 1, 4};\n',
...           'Circle(8) = {4, 1, 5};\n',
```

```
...                  'Circle(9) = {5, 1, 2};\n',
...                  'Line Loop(10) = {6, 7, 8, 9} ;\n',
...                  'Plane Surface(11) = {10};\n']

>>> import tempfile
>>> (f, geomName) = tempfile.mkstemp('.geo')
>>> file = open(geomName, 'w')
>>> file.writelines(lines)
>>> file.close()
>>> import os
>>> os.close(f)
```

The temporary file created above is used by gmsh to mesh the geometrically defined region.

```
>>> import sys
>>> if sys.platform == 'win32':
...     meshName = 'tmp.msh'
... else:
...     (f, meshName) = tempfile.mkstemp('.msh')
>>> os.system('gmsh ' + geomName + ' -2 -v 0 -format msh -o ' + meshName)
0
>>> if sys.platform != 'win32':
...     os.close(f)
>>> os.remove(geomName)
```

The mesh created by gmsh is then imported into FiPy using the `GmshImporter2D` object.

```
>>> from fipy.meshes.gmshImport import GmshImporter2D
>>> mesh = GmshImporter2D(meshName)
>>> os.remove(meshName)
```

Using this mesh, we can construct a solution variable

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phi = CellVariable(name = "solution variable",
...                    mesh = mesh,
...                    value = 0)
```

We can now create a viewer to see the mesh (only the `Gist2DViewer` is capable of displaying variables on this sort of irregular mesh)

```
>>> viewer = None
>>> if __name__ == '__main__':
...     try:
...         from fipy.viewers.gistViewer.gist2DViewer import Gist2DViewer
...         viewer = Gist2DViewer(vars=phi,
...                               limits={'datamin': -1, 'datamax': 1.})
...         viewer.plotMesh()
...         raw_input("Irregular circular mesh. Press <return> to proceed...")
...     except:
...         print "Unable to create a Gist2DViewer"
```
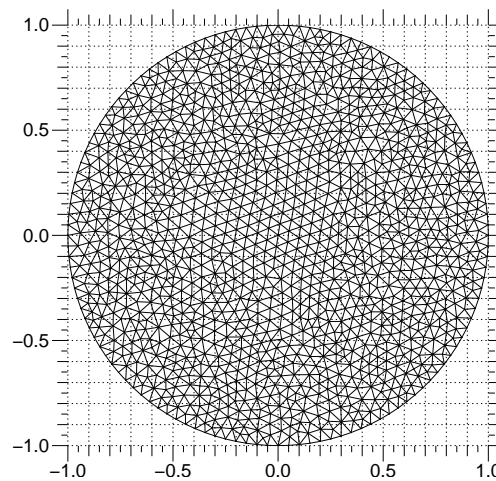
We set up a transient diffusion equation

```
>>> D = 1.
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> eq = TransientTerm() == ImplicitDiffusionTerm(coeff=D)
```

The following line extracts the x coordinate values on the exterior faces. These are used as the boundary condition fixed values.

```
>>> from fipy.tools import numerix
>>> exteriorXcoords = numerix.take(mesh.getFaceCenters()[...,0],
...                                mesh.getExteriorFaces())
```

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> BCs = (FixedValue(faces=mesh.getExteriorFaces(), value=exteriorXcoords),)
```

We first step through the transient problem

```
>>> timeStepDuration = 10 * 0.9 * cellSize**2 / (2 * D)
>>> steps = 10
>>> for step in range(steps):
...     eq.solve(var=phi,
...              boundaryConditions=BCs,
...              dt=timeStepDuration)
...     if viewer is not None:
...         viewer.plot()
```
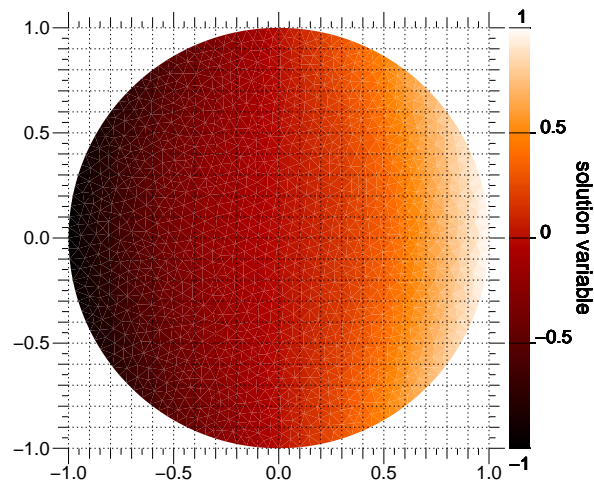
If we wanted to plot or analyze the results of this calculation with another application, we could export tab-separated-values with

```
from fipy.viewers.tsvViewer import TSVViewer
TSVViewer(vars=(phi, phi.getGrad())).plot(filename="myTSV.tsv")
```

x	y	solution variable	solution variable_grad_x	solution variable_grad_y
0.975559734792414	0.0755414402612554	0.964844363287199	-0.229687917881182	0.00757854476106331
0.0442864953037566	0.79191893162384	0.0375859836421991	-0.773936613923853	-0.205560697612547
0.0246775505084069	0.771959648896982	0.020853932412869	-0.723540342405813	-0.182589694334729
0.223345558247991	-0.807931073108895	0.203035857140125	-0.777466238738658	0.0401235242511506
-0.00726763301939488	-0.775978916110686	-0.00412895434496877	-0.650055516507232	-0.183112882869288
-0.0220279064527904	-0.187563765977912	-0.012771874945585	-0.35707168379437	-0.056072788439713
0.111223320911545	-0.679586798311355	0.0911595298310758	-0.613455176718145	0.0256182541329463
-0.78996770899909	-0.0173672729866294	-0.693887874335319	-1.00671109050419	-0.127611490372511
-0.703545986179876	-0.435813500559859	-0.635004192597412	-0.896203033957194	-0.00855563518923689
0.888641841567831	-0.408558914368324	0.877939107374768	-0.32195762184087	-0.22696791637322
0.38212257821916	-0.51732949653553	0.292889724306196	-0.854466141879776	0.199715815696975
-0.359068256998365	0.757882581524374	-0.323541041763627	-0.870534227755687	0.0792631912863636
-0.459673905457569	-0.701526587772079	-0.417577664032421	-0.725460726303266	-0.119132299176163
-0.338256179134518	-0.523565732643067	-0.254030052182524	-0.923505840608445	-0.192224240688976
0.87498754712638	0.174119064688993	0.836057900916614	-1.11590500805745	-0.211010116496191
-0.484106960369249	0.0705987421869745	-0.319827850867342	-0.867894407968447	0.051246727010685
-0.0221203060940465	-0.216026820080053	-0.0152729438559779	-0.341246696530392	-0.0538476142281317

The values are listed at the `Cell` centers. Particularly for irregular meshes, no specific ordering should be relied upon. Vector quantities are listed in multiple columns, one for each mesh dimension.

This problem again has an analytical solution that depends on the error function, but it's a bit more complicated due to the varying boundary conditions and the different horizontal diffusion length at different vertical positions

```
>>> x, y = mesh.getCellCenters()[...,0], mesh.getCellCenters()[...,1]
>>> t = timeStepDuration * steps

>>> phiAnalytical = CellVariable(name="analytical value",
...                              mesh=mesh)


>>> from fipy.tools.numerix import sqrt, arcsin, cos
>>> x0 = radius * cos(arcsin(y))
>>> try:
```

```
...        from scipy.special import erf
...        phiAnalytical.setValue(x0 * (erf((x0+x) / (2 * sqrt(D * t)))
...                                    - erf((x0-x) / (2 * sqrt(D * t)))))
... except ImportError:
...        print "The SciPy library is not available to test the solution to \
... the transient diffusion equation"

>>> print phi.allclose(phiAnalytical, atol = 6e-2)
1


>>> if __name__ == '__main__':
...        raw_input("Transient diffusion. Press <return> to proceed...")
```

As in the earlier examples, we can also directly solve the steady-state diffusion problem.

```
>>> ImplicitDiffusionTerm(coeff=D).solve(var=phi,
...                                    boundaryConditions=BCs)
```

The values at the elements should be equal to their x coordinate

```
>>> print phi.allclose(mesh.getCellCenters()[...,0], atol = 0.02)
1
```

Display the results if run as a script.

```
>>> if viewer is not None:
...        viewer.plot()
...        raw_input("Steady-state diffusion. Press <return> to proceed...")
```
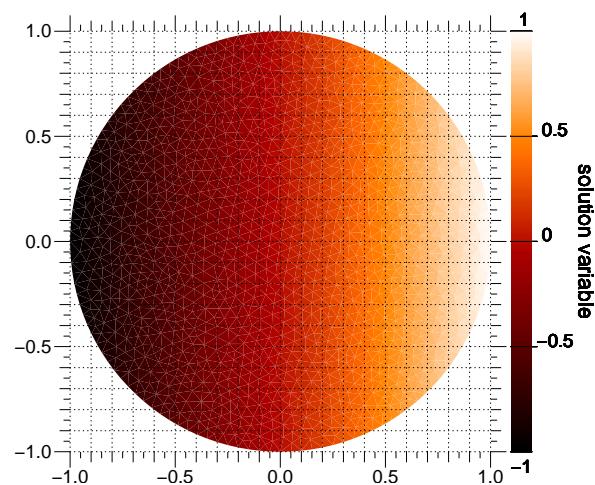

7.4   Module `examples.diffusion.electrostatics`

The Poisson equation is a particular example of the steady-state diffusion equation. We examine a few cases in one dimension.

```
>>> nx = 200
>>> dx = 0.01
>>> L = nx * dx
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = dx, nx = nx)
```

Given the electrostatic potential ϕ,

```
>>> from fipy.variables.cellVariable import CellVariable
>>> potential = CellVariable(mesh=mesh, name='potential', value=0.)
```

the permittivity ϵ,

```
>>> permittivity = 1
```

the concentration C_j of the jth component with valence z_j (we consider only a single component C_{e}^- with valence with $z_{\text{e}^-} = -1$)

```
>>> electrons = CellVariable(mesh=mesh, name='e-')
>>> electrons.valence = -1
```

and the charge density ρ,

```
>>> charge = electrons * electrons.valence
>>> charge.name = "charge"
```

The dimensionless Poisson equation is

$$\nabla \cdot (\epsilon \nabla \phi) = -\rho = -\sum_{j=1}^{n} z_j C_j$$

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> potential.equation = ImplicitDiffusionTerm(coeff = permittivity) \
...                          + charge == 0
```

Because this equation admits an infinite number of potential profiles, we must constrain the solution by fixing the potential at one point:

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> bcs = (FixedValue(faces=mesh.getFacesLeft(), value=0),)
```

First, we obtain a uniform charge distribution by setting a uniform concentration of electrons $C_{\text{e}^-} = 1$.

```
>>> electrons.setValue(1.)
```

and we solve for the electrostatic potential

```
>>> potential.equation.solve(var=potential,
...                          boundaryConditions=bcs)
```

This problem has the analytical solution

$$\psi(x) = \frac{x^2}{2} - 2x$$

```
>>> x = mesh.getCellCenters()[:,0]
>>> analytical = CellVariable(mesh=mesh, name="analytical solution",
...                           value=(x**2)/2 - 2*x)
```

which has been satisifactorily obtained

```
>>> print potential.allclose(analytical, rtol = 2e-5, atol = 2e-5)
1
```

If we are running the example interactively, we view the result

```
>>> if __name__ == '__main__':
...     import fipy.viewers
...     viewer = fipy.viewers.make(vars=(charge, potential, analytical))
...     viewer.plot()
...     raw_input("Press any key to continue...")
```
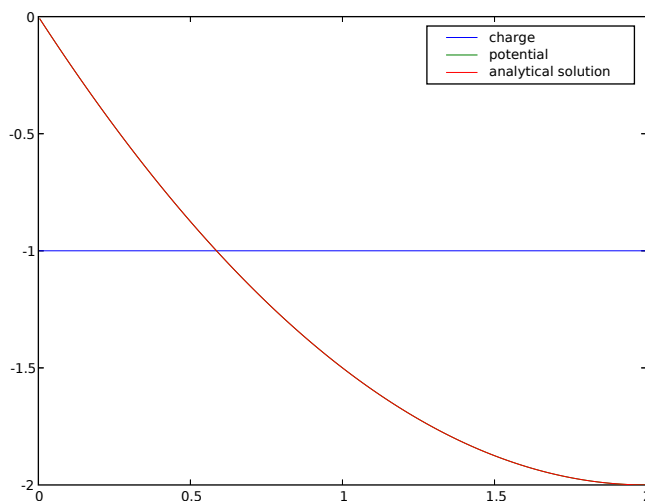

Next, we segregate all of the electrons to right side of the domain

$$C_{e^-} = \begin{cases} 0 & \text{for } x \leq L/2, \\ 1 & \text{for } x > L/2. \end{cases}$$

```
>>> x = mesh.getCellCenters()[...,0]
>>> electrons.setValue(0.)
>>> electrons.setValue(1., where=x > L / 2.)
```

and again solve for the electrostatic potential

```
>>> potential.equation.solve(var=potential,
...                          boundaryConditions=bcs)
```

which now has the analytical solution

$$\psi(x) = \begin{cases} -x & \text{for } x \leq L/2, \\ \frac{(x-1)^2}{2} - x & \text{for } x > L/2. \end{cases}$$

```
>>> analytical.setValue(-x)
>>> analytical.setValue(((x-1)**2)/2 - x, where=x > L/2)

>>> print potential.allclose(analytical, rtol = 2e-5, atol = 2e-5)
1
```

and again view the result

```
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("Press any key to continue...")
```
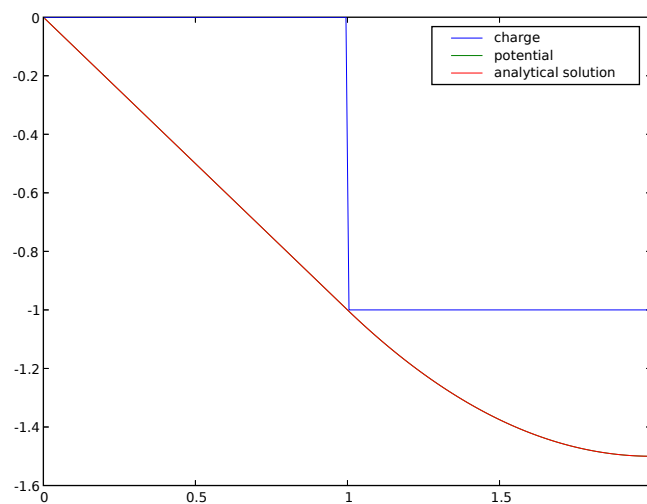

Finally, we segregate all of the electrons to the left side of the domain

$$C_{e^-} = \begin{cases} 1 & \text{for } x \leq L/2, \\ 0 & \text{for } x > L/2. \end{cases}$$

```
>>> electrons.setValue(1.)
>>> electrons.setValue(0., where=x > L / 2.)
```

and again solve for the electrostatic potential

```
>>> potential.equation.solve(var=potential,
...                          boundaryConditions=bcs)
```

which has the analytical solution

$$\psi(x) = \begin{cases} \frac{x^2}{2} - x & \text{for } x \leq L/2, \\ -\frac{1}{2} & \text{for } x > L/2. \end{cases}$$

We again verify that the correct equilibrium is attained

```
>>> analytical.setValue((x**2)/2 - x)
>>> analytical.setValue(-0.5, where=x > L / 2)
```

```
>>> print potential.allclose(analytical, rtol = 2e-5, atol = 2e-5)
1
```

and once again view the result

```
>>> if __name__ == '__main__':
...     viewer.plot()
```
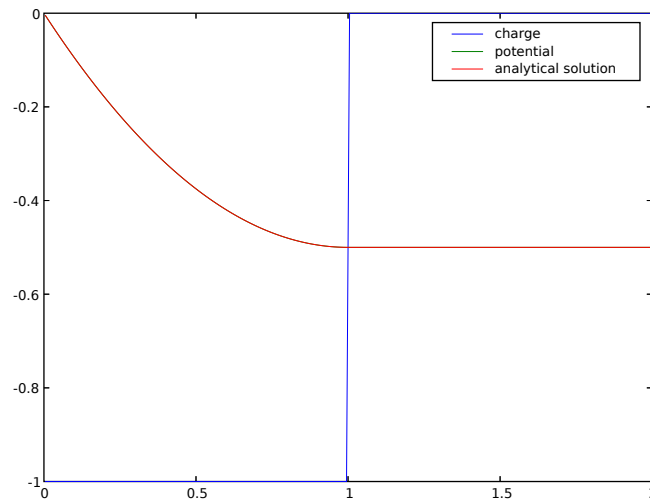

7.5 Module `examples.diffusion.nthOrder.input4thOrder1D`

This example uses the `ImplicitDiffusionTerm` class to solve the equation

$$\frac{\partial^4 \phi}{\partial x^4} = 0$$

on a 1D mesh of length

```
>>> L = 1000.
```

We create an appropriate mesh

```
>>> nx = 1000
>>> dx = L / nx
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx=dx, nx=nx)
```

and initialize the solution variable to 0

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh=mesh, name='solution variable')
```

For this problem, we impose the boundary conditions:

$$\phi = \alpha_1 \quad \text{at } x = 0$$

$$\frac{\partial \phi}{\partial x} = \alpha_2 \quad \text{at } x = L$$

$$\frac{\partial^2 \phi}{\partial x^2} = \alpha_3 \quad \text{at } x = 0$$

$$\frac{\partial^3 \phi}{\partial x^3} = \alpha_4 \quad \text{at } x = L.$$

or

```
>>> alpha1 = 2.
>>> alpha2 = 1.
>>> alpha3 = 4.
>>> alpha4 = -3.


>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> from fipy.boundaryConditions.nthOrderBoundaryCondition \
...       import NthOrderBoundaryCondition
>>> BCs = (FixedValue(faces=mesh.getFacesLeft(), value=alpha1),
...        FixedFlux(faces=mesh.getFacesRight(), value=alpha2),
...        NthOrderBoundaryCondition(faces=mesh.getFacesLeft(), value=alpha3, order=2),
...        NthOrderBoundaryCondition(faces=mesh.getFacesRight(), value=alpha4, order=3))
```

We initialize the steady-state equation

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> eq = ImplicitDiffusionTerm(coeff=(1, 1)) == 0
```

and use the `LinearLUSolver` for stability.

```
>>> from fipy.solvers.linearLUSolver import LinearLUSolver
```

We perform one implicit timestep to achieve steady state

```
>>> eq.solve(var=var,
...          boundaryConditions=BCs,
...          solver=LinearLUSolver(tolerance=1e-11))
```

The analytical solution is:

$$\phi = \frac{\alpha_4}{6}x^3 + \frac{\alpha_3}{2}x^2 + \left(\alpha_2 - \frac{\alpha_4}{2}L^2 - \alpha_3 L\right)x + \alpha_1$$

or

```
>>> analytical = CellVariable(mesh=mesh, name='analytical value')
>>> x = mesh.getCellCenters()[:,0]
>>> analytical.setValue(alpha4 / 6. * x**3 + alpha3 / 2. * x**2 + \
...                     (alpha2 - alpha4 / 2. * L**2 - alpha3 * L) * x + alpha1)
>>> print var.allclose(analytical, rtol=1e-4)
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers import make
...     viewer = make(vars=(var, analytical))
...     viewer.plot()
```
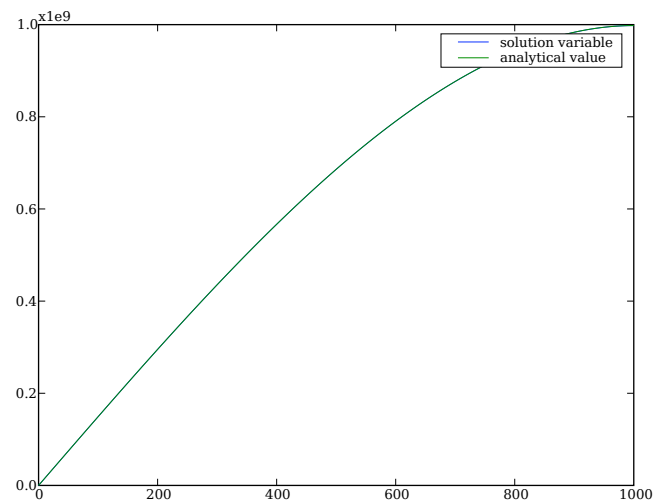
Chapter 8

Convection Examples

8.1   Module `examples.convection.exponential1D.input`

This example solves the steady-state convection-diffusion equation given by:

$$\nabla \cdot (D\nabla\phi + \vec{u}\phi) = 0$$

with coefficients $D = 1$ and $\vec{u} = (10,)$, or

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,)
```

We define a 1D mesh

```
>>> L = 10.
>>> nx = 10
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx=L / nx, nx=nx)
```

and impose the boundary conditions

$$\phi = \begin{cases} 0 & \text{at } x = 0, \\ 1 & \text{at } x = L, \end{cases}$$

or

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> boundaryConditions = (
...       FixedValue(faces=mesh.getFacesLeft(), value=valueLeft),
...       FixedValue(faces=mesh.getFacesRight(), value=valueRight),
...       )
```

The solution variable is initialized to `valueLeft`:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh=mesh, name = "variable")
```

The equation is created with the `ImplicitDiffusionTerm` and `ExponentialConvectionTerm`. The scheme used by the convection term needs to calculate a Peclet number and thus the diffusion term instance must be passed to the convection term.

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> diffTerm = ImplicitDiffusionTerm(coeff=diffCoeff)

>>> from fipy.terms.exponentialConvectionTerm \
...     import ExponentialConvectionTerm
>>> eq = diffTerm + ExponentialConvectionTerm(coeff=convCoeff,
...                                            diffusionTerm=diffTerm)
```

More details of the benefits and drawbacks of each type of convection term can be found in Section 3.5 "Numerical Schemes".

Essentially, the `ExponentialConvectionTerm` and `PowerLawConvectionTerm` will both handle most types of convection-diffusion cases, with the `PowerLawConvectionTerm` being more efficient.

We solve the equation

```
>>> eq.solve(var=var, boundaryConditions=boundaryConditions)
```

and test the solution against the analytical result

$$\phi = \frac{1 - \exp(-u_x x/D)}{1 - \exp(-u_x L/D)}$$

or

```
>>> axis = 0
>>> x = mesh.getCellCenters()[:,axis]
>>> from fipy.tools.numerix import exp
>>> CC = 1. - exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
>>> print var.allclose(analyticalArray)
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers import make
...     viewer = make(vars=var)
...     viewer.plot()
```

8.2   Module `examples.convection.exponential1DSource.input`

 Like `examples/diffusion/convection/exponential1D/input.py` this example solves a steady-state convection-diffusion equation, but adds a constant source, $S_0 = 1$, such that

$$\nabla \cdot (D\nabla\phi + \vec{u}\phi) + S_0 = 0.$$

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,)
>>> sourceCoeff = 1.
```

We define a 1D mesh

```
>>> nx = 1000
>>> L = 10.
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx=L / 1000, nx=nx)
```

and impose the boundary conditions

$$\phi = \begin{cases} 0 & \text{at } x = 0, \\ 1 & \text{at } x = L, \end{cases}$$

or

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> boundaryConditions = (
...      FixedValue(faces=mesh.getFacesRight(), value=valueRight),
...      FixedValue(faces=mesh.getFacesLeft(), value=valueLeft),
...      )
```

The solution variable is initialized to `valueLeft`:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(name="variable", mesh=mesh)
```

We define the convection-diffusion equation with source

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> from fipy.terms.exponentialConvectionTerm \
...      import ExponentialConvectionTerm
>>> diffTerm = ImplicitDiffusionTerm(coeff=diffCoeff)
>>> eq = diffTerm \
...      + ExponentialConvectionTerm(coeff=convCoeff,
...                                  diffusionTerm=diffTerm) \
...      + sourceCoeff

>>> from fipy.solvers.linearLUSolver import LinearLUSolver
>>> eq.solve(var = var,
...          boundaryConditions = boundaryConditions,
...          solver = LinearLUSolver(tolerance = 1.e-15))
```

and test the solution against the analytical result:

$$\phi = -\frac{S_0 x}{u_x} + \left(1 + \frac{S_0 x}{u_x}\right) \frac{1 - \exp(-u_x x/D)}{1 - \exp(-u_x L/D)}$$

or

```
>>> axis = 0
>>> x = mesh.getCellCenters()[:,axis]
>>> AA = -sourceCoeff * x / convCoeff[axis]
>>> BB = 1. + sourceCoeff * L / convCoeff[axis]
>>> from fipy.tools.numerix import exp
>>> CC = 1. - exp(-convCoeff[axis] * x / diffCoeff)
```

```
>>> DD = 1. - exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = AA + BB * CC / DD
>>> print var.allclose(analyticalArray, rtol=1e-4, atol=1e-4)
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers import make
...     viewer = make(vars=var)
...     viewer.plot()
```

Chapter 9

Phase Field Examples

The phase field method is a "diffuse interface" technique for modeling phase transformations and interface motion. Several good review articles have been written on the subject [2, 3, 4].

9.1 Module `examples.phase.simple.input`

To run this example from the base FiPy directory, type `examples/phase/simple/input.py` at the command line. A viewer object should appear and, after being prompted to step through the different examples, the word `finished` in the terminal.

This example takes the user through assembling a simple problem with FiPy. It describes a steady 1D phase field problem with no-flux boundary conditions such that,

$$\frac{1}{M_\phi}\frac{\partial \phi}{\partial t} = \kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \tag{9.1}$$

For solidification problems, the Helmholtz free energy is frequently given by

$$f(\phi, T) = \frac{W}{2}g(\phi) + L_v \frac{T - T_M}{T_M}p(\phi).$$

where W is the double-well barrier height between phases, L_v is the latent heat, T is the temperature, and T_M is the melting point. One possible choice for the double-well function is

$$g(\phi) = \phi^2(1 - \phi)^2$$

and for the interpolation function is

$$p(\phi) = \phi^3(6\phi^2 - 15\phi + 10).$$

We create a 1D solution mesh

```
>>> L = 1.
>>> nx = 400
>>> dx = L / nx


>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = dx, nx = nx)
```

We create the phase field variable

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phase = CellVariable(name = "phase",
...                                mesh = mesh)
```

and set a step-function initial condition

$$\phi = \begin{cases} 1 & \text{for } x \leq L/2 \\ 0 & \text{for } x > L/2 \end{cases} \quad \text{at } t = 0$$

```
>>> x = mesh.getCellCenters()[...,0]
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

If we are running interactively, we'll want a viewer to see the results

```
>>> if __name__ == '__main__':
...       import fipy.viewers
...       viewer = fipy.viewers.make(vars = (phase,))
...       viewer.plot()
...       raw_input("Initial condition. Press <return> to proceed...")
```
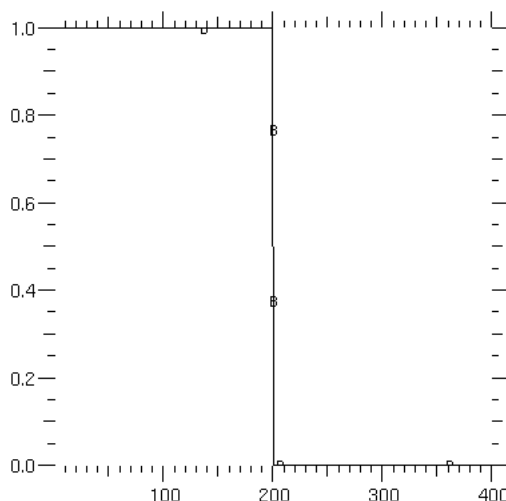

We choose the parameter values,

```
>>> kappa = 0.0025
>>> W = 1.
>>> Lv = 1.
>>> Tm = 1.
>>> T = Tm
>>> enthalpy = Lv * (T - Tm) / Tm
```

We build the equation by assembling the appropriate terms. Since, with $T = T_M$ we are interested in a steady-state solution, we omit the transient term $(1/M_\phi)\frac{\partial \phi}{\partial t}$. The analytical solution for this steady-state phase field problem, in an infinite domain, is

$$\phi = \frac{1}{2}\left[1 - \tanh\frac{x - L/2}{2\sqrt{\kappa/W}}\right] \qquad (9.2)$$

or

```
>>> x = mesh.getCellCenters()[:,0]
>>> from fipy.tools.numerix import tanh, sqrt
>>> analyticalArray = 0.5*(1 - tanh((x - L/2)/(2*sqrt(kappa/W))))
```

We treat the diffusion term $\kappa_\phi \nabla^2 \phi$ implicitly,

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> diffusionTerm = ImplicitDiffusionTerm(coeff = kappa)
```

Note

"Diffusion" in FiPy is not limited to the movement of atoms, but rather refers to the spontaneous spreading of any quantity (e.g., solute, temperature, or in this case "phase") by flow "down" the gradient of that quantity.

The source term is

$$S = -\frac{\partial f}{\partial \phi} = \frac{W}{2} g'(\phi) + L \frac{T - T_M}{T_M} p'(\phi)$$

$$= -\left[W\phi(1-\phi)(1-2\phi) + L\frac{T-T_M}{T_M} 30\phi^2(1-\phi)^2 \right]$$

$$= m_\phi \phi(1-\phi)$$

where $m_\phi \equiv -[W(1-2\phi) + 30\phi(1-\phi)L\frac{T-T_M}{T_M}]$. The simplest approach is to add this source explicitly

```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> S0 = mPhi * phase * (1 - phase)
>>> eq = S0 + diffusionTerm
```

After solving this equation

```
>>> eq.solve(var = phase)
```

we obtain the surprising result that ϕ is zero everywhere.

```
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("Fully explicit source. Press <return> to proceed...")
>>> print phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4)
0
```
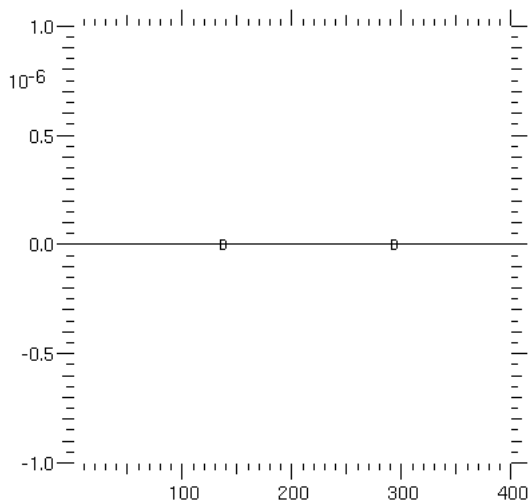
On inspection, we can see that this occurs because, for our step-function initial condition, $m_\phi = 0$ everwhere, hence we are actually only solving the simple implicit diffusion equation $\kappa_\phi \nabla^2 \phi = 0$, which has exactly the uninteresting solution we obtained.

The resolution to this problem is to apply relaxation to obtain the desired answer, i.e., the solution is allowed to relax in time from the initial condition to the desired equilibrium solution. To do so, we reintroduce the transient term from Equation (9.1)

```
>>> from fipy.terms.transientTerm import TransientTerm
>>> eq = TransientTerm() == diffusionTerm + S0

>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)

>>> for i in range(13):
...         eq.solve(var = phase)
...         if __name__ == '__main__':
...             viewer.plot()
>>> if __name__ == '__main__':
...         raw_input("Relaxation, explicit. Press <return> to proceed...")
```
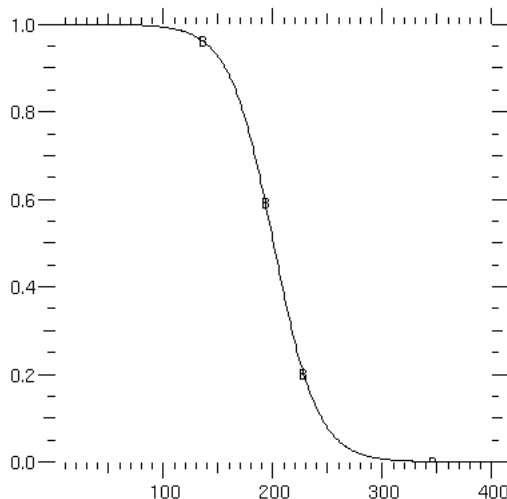

After 13 time steps, the solution has converged to the analytical solution

```
>>> print phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4)
1
```

Note

The solution is only found accurate to $\approx 4.3 \times 10^{-5}$ because the infinite-domain analytical solution (9.2) is not an exact representation for the solution in a finite domain of length L.

Setting fixed-value boundary conditions of 1 and 0 would still require the relaxation method with the fully explicit source.

Solution performance can be improved if we exploit the dependence of the source on ϕ. By doing so, we can make the source semi-implicit, improving the rate of convergence over the fully explicit approach. The source can only be semi-implicit because we employ sparse linear algebra routines to solve the PDEs, i.e.,

there is no fully implicit way to represent a term like ϕ^4 in the linear set of equations $\mathsf{M}\vec{\phi} - \vec{b} = 0$. By linearizing a source as $S = S_0 - S_1\phi$, we make it more implicit by adding the coefficient S_1 to the matrix diagonal. For numerical stability, this linear coefficient must never be negative.

There are an infinite number of choices for this linearization, but many do not converge very well. One choice is that used by Ryo Kobayashi:

```
>>> from fipy.terms.implicitSourceTerm import ImplicitSourceTerm
>>> S0 = mPhi * phase * (mPhi > 0)
>>> S1 = mPhi * ((mPhi < 0) - phase)
>>> implicitSource = ImplicitSourceTerm(coeff = S1)
>>> eq = diffusionTerm + S0 + implicitSource
```

> **Note**
>
> Because `mPhi` is a variable field, the quantities (`mPhi` > 0) and (`mPhi` < 0) evaluate to variable *fields* of ones and zeroes, instead of simple boolean values.

This expression converges to the same value given by the explicit relaxation approach, but in only 8 sweeps (note that because there is no transient term, these sweeps are not time steps, but rather repeated iterations at the same time step to reach a converged solution).

> **Note**
>
> We use `solve()` instead of `sweep()` because we don't care about the residual. Either function would work, but `solve()` is a bit faster.

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)

>>> for i in range(8):
...     eq.solve(var = phase)
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("Kobayashi, semi-implicit. Press <return> to proceed...")
>>> print phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4)
1
```

In general, the best convergence is obtained when the linearization gives a good representation of the relationship between the source and the dependent variable. The best practical advice is to perform a Taylor expansion of the source about the previous value of the dependent variable such that $S = S_{\mathrm{old}} + \left.\frac{\partial S}{\partial \phi}\right|_{\mathrm{old}} (\phi - \phi_{\mathrm{old}}) = (S - \frac{\partial S}{\partial \phi}\phi)_{\mathrm{old}} + \left.\frac{\partial S}{\partial \phi}\right|_{\mathrm{old}} \phi$. Now, if our source term is represented by $S = S_0 + S_1\phi$, then $S_1 = \left.\frac{\partial S}{\partial \phi}\right|_{\mathrm{old}}$ and $S_0 = (S - \frac{\partial S}{\partial \phi}\phi)_{\mathrm{old}} = S_{\mathrm{old}} - S_1\phi_{\mathrm{old}}$. In this way, the linearized source will be tangent to the curve of the actual source as a function of the dependent variable.

For our source, $S = m_\phi\phi(1 - \phi)$,

$$\frac{\partial S}{\partial \phi} = \frac{\partial m_\phi}{\partial \phi}\phi(1 - \phi) + m_\phi(1 - 2\phi)$$

and

$$\frac{\partial m_\phi}{\partial \phi} = 2W - 30(1 - 2\phi)L\frac{T - T_M}{T_M},$$

or

```
>>> dmPhidPhi = 2 * W - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase * (S1 < 0)
>>> implicitSource = ImplicitSourceTerm(coeff = S1 * (S1 < 0))
>>> eq = diffusionTerm + S0 + implicitSource
```

Using this scheme, where the coefficient of the implicit source term is tangent to the source, we reach convergence in only 5 sweeps

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)


>>> for i in range(5):
...     eq.solve(var = phase)
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("Tangent, semi-implicit. Press <return> to proceed...")
>>> print phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4)
1
```

Although, for this simple problem, there is no appreciable difference in run-time between the fully explicit source and the optimized semi-implicit source, the benefit of 60% fewer sweeps should be obvious for larger systems and longer iterations.

———————————————————

This example has focused on just the region of the phase field interface in equilibrium. Problems of interest, though, usually involve the dynamics of one phase transforming to another. To that end, let us recast the problem using physical parameters and dimensions. We'll need a new mesh

```
>>> nx = 400
>>> dx = 5e-6 # cm
>>> L = nx * dx


>>> mesh = Grid1D(dx = dx, nx = nx)
```

and thus must redeclare ϕ on the new mesh

```
>>> phase = CellVariable(name="phase",
...                      mesh=mesh,
...                      hasOld=1)
>>> x = mesh.getCellCenters()[...,0]
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

We choose the parameter values appropriate for nickel, given in [28]

```
>>> Lv = 2350 # J / cm**3
>>> Tm = 1728. # K
>>> from fipy.variables.variable import Variable
>>> T = Variable(value=Tm)
>>> enthalpy = Lv * (T - Tm) / Tm # J / cm**3
```

The parameters of the phase field model can be related to the surface energy σ and the interfacial thickness δ by

$$\kappa = 6\sigma\delta$$
$$W = \frac{6\sigma}{\delta}$$
$$M_\phi = \frac{T_m\beta}{6L\delta}.$$

We take $\delta \approx \Delta x$.

```
>>> delta = 1.5 * dx
>>> sigma = 3.7e-5 # J / cm**2
>>> beta = 0.33 # cm / (K s)
>>> kappa = 6 * sigma * delta # J / cm
>>> W = 6 * sigma / delta # J / cm**3
>>> Mphi = Tm * beta / (6. * Lv * delta) # cm**3 / (J s)

>>> analyticalArray = CellVariable(name="tanh", mesh=mesh,
...                                value=0.5 * (1 - tanh((x - (L / 2. + L / 10.))
...                                                      / (2 * delta))))
```

and make a new viewer

```
>>> if __name__ == '__main__':
...     viewer2 = fipy.viewers.make(vars = (phase, analyticalArray))
...     viewer2.plot()
```

Now we can redefine the transient phase field equation, using the optimal form of the source term shown above

```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> dmPhidPhi = 2 * W - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase * (S1 < 0)
>>> eq = TransientTerm(coeff=1/Mphi) == ImplicitDiffusionTerm(coeff=kappa) \
...                       + S0 + ImplicitSourceTerm(coeff = S1 * (S1 < 0))
```

In order to separate the effect of forming the phase field interface from the kinetics of moving it, we first equilibrate at the melting point. We now use the "`sweep()`" method instead of "`solve()`" because we require the residual.

```
>>> timeStep = 1e-6
>>> for i in range(10):
...     phase.updateOld()
...     res = 1e+10
...     while res > 1e-5:
...         res = eq.sweep(var=phase, dt=timeStep)
>>> if __name__ == '__main__':
...     viewer2.plot()
```

and then quench by 1 K

```
>>> T.setValue(T() - 1)
```

In order to have a stable numerical solution, the interface must not move more than one grid point per time step, we thus set the timestep according to the grid spacing Δx, the linear kinetic coefficient β, and the undercooling $|T_m - T|$
Again we use the "`sweep()`" method as a replacement for "`solve()`".

```
>>> velocity = beta * abs(Tm - T()) # cm / s
>>> timeStep = .1 * dx / velocity # s
>>> elapsed = 0
>>> while elapsed < 0.1 * L / velocity:
...     phase.updateOld()
...     res = 1e+10
...     while res > 1e-5:
...         res = eq.sweep(var=phase, dt=timeStep)
...     elapsed += timeStep
...     if __name__ == '__main__':
...         viewer2.plot()
```

A hyperbolic tangent is not an exact steady-state solution given the quintic polynomial we chose for the `p()` function, but it gives a reasonable approximation.

```
>>> print phase.allclose(analyticalArray, rtol = 5, atol = 2e-3)
1
```

If we had made another common choice of $p(\phi) = \phi^2(3 - 2\phi)$, we would have found much better agreement, as that case does give an exact tanh solution in steady state. If SciPy is available, another way to compare against the expected result is to do a least-squared fit to determine the interface velocity and thickness

```
>>> try:
...     def tanhResiduals(p, y, x, t):
...         V, d = p
...         return y - 0.5 * (1 - tanh((x - V * t - L / 2.) / (2*d)))
...     from scipy.optimize import leastsq
...     x =  mesh.getCellCenters()[...,0]
...     (V_fit, d_fit), msg = leastsq(tanhResiduals, [L/2., delta],
...                             args=(phase(), x, elapsed))
... except ImportError:
...     V_fit = d_fit = 0
...     print "The SciPy library is unavailable to fit the interface \
... thickness and velocity"

>>> print abs(1 - V_fit / velocity) < 3e-2
True
>>> print abs(1 - d_fit / delta) < 2e-2
True


>>> if __name__ == '__main__':
...     raw_input("Dimensional, semi-implicit. Press <return> to proceed...")
```
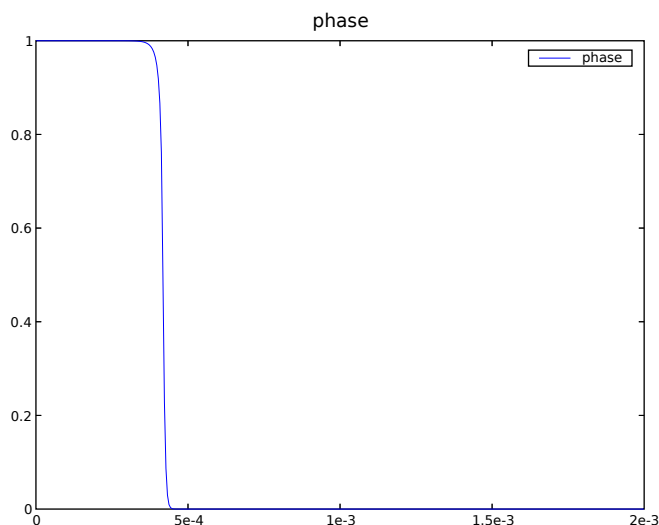
9.2   Module `examples.phase.binary`

It is straightforward to extend a phase field model to include binary alloys.   As in `examples.phase.simple.input`, we will examine a 1D problem

```
>>> nx = 400
>>> dx = 5e-6 # cm
>>> L = nx * dx
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx=dx, nx=nx)
```

The Helmholtz free energy functional can be written as the integral [2, 4, 29]

$$\mathcal{F}\left(\phi, C, T\right) = \int_{\mathcal{V}} \left\{ f(\phi, C, T) + \frac{\kappa_{\phi}}{2} |\nabla\phi|^2 + \frac{\kappa_C}{2} |\nabla C|^2 \right\} dV$$

over the volume $\mathcal{V}$ as a function of phase[1] ϕ

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phase = CellVariable(name="phase", mesh=mesh, hasOld=1)
```

composition C

```
>>> C = CellVariable(name="composition", mesh=mesh, hasOld=1)
```

and temperature[2] T

```
>>> from fipy.variables.variable import Variable
>>> T = Variable(name="temperature")
```

---

[1]We will find that we need to "sweep" this non-linear problem (see *e.g.* the composition-dependent diffusivity example in `examples.diffusion.mesh1D`), so we declare ϕ and C to retain an "old" value.

[2]we are going to want to examine different temperatures in this example, so we declare T as a `Variable`

Frequently, the gradient energy term in concentration is ignored and we can derive governing equations

$$\frac{\partial \phi}{\partial t} = M_\phi \left(\kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \right) \tag{9.3}$$

for phase and

$$\frac{\partial C}{\partial t} = \nabla \cdot \left(M_C \nabla \frac{\partial f}{\partial C} \right) \tag{9.4}$$

for solute.

The free energy density $f(\phi, C, T)$ can be constructed in many different ways. One approach is to construct free energy densities for each of the pure compoonents, as functions of phase, $e.g.$

$$f_A(\phi, T) = p(\phi) f_A^S(T) + (1 - p(\phi)) f_A^L(T) + \frac{W_A}{2} g(\phi)$$

where $f_A^L(T)$, $f_B^L(T)$, $f_A^S(T)$, and $f_B^S(T)$ are the free energy densities of the pure components. There are a variety of choices for the interpolation function $p(\phi)$ and the barrier function $g(\phi)$, such as those shown in `examples.phase.simple.input`

```
>>> def p(phi):
...     return phi**3 * (6 * phi**2 - 15 * phi + 10)


>>> def g(phi):
...     return (phi * (1 - phi))**2
```

The desired thermodynamic model can then be applied to obtain $f(\phi, C, T)$, such as for a regular solution,

$$
\begin{aligned}
f(\phi, C, T) = {} & (1 - C) f_A(\phi, T) + C f_B(\phi, T) \\
& + RT \left[(1 - C) \ln(1 - C) + C \ln C \right] + C(1 - C) \left[\Omega_S p(\phi) + \Omega_L \left(1 - p(\phi) \right) \right]
\end{aligned}
$$

where

```
>>> R = 8.314 # J / (mol K)
```

is the gas constant and Ω_S and Ω_L are the regular solution interaction parameters for solid and liquid. Another approach is useful when the free energy densities $f^L(C, T)$ and $f^S(C, T)$ of the alloy in the solid and liquid phases are known. This might be the case when the two different phases have different thermodynamic models or when one or both is obtained from a Calphad code. In this case, we can construct

$$f(\phi, C, T) = p(\phi) f^S(C, T) + (1 - p(\phi)) f^L(C, T) + \left[(1 - C) \frac{W_A}{2} + C \frac{W_B}{2} \right] g(\phi).$$

When the thermodynamic models are the same in both phases, both approaches should yield the same result. We choose the first approach and make the simplifying assumptions of an ideal solution and that

$$f_A^L(T) = 0$$
$$f_A^S(T) - f_A^L(T) = \frac{L_A \left(T - T_M^A \right)}{T_M^A}$$

and likewise for component B.

```
>>> LA = 2350. # J / cm**3
>>> LB = 1728. # J / cm**3
>>> TmA = 1728. # K
>>> TmB = 1358. # K

>>> enthalpyA = LA * (T - TmA) / TmA
>>> enthalpyB = LB * (T - TmB) / TmB
```

This relates the difference between the free energy densities of the pure solid and pure liquid phases to the latent heat L_A and the pure component melting point T_M^A, such that

$$f_A(\phi, T) = \frac{L_A \left(T - T_M^A\right)}{T_M^A} p(\phi) + \frac{W_A}{2} g(\phi).$$

With these assumptions

$$\frac{\partial f}{\partial \phi} = (1 - C)\frac{\partial f_A}{\partial \phi} + C\frac{\partial f_A}{\partial \phi}$$

$$= \left\{(1 - C)\frac{L_A \left(T - T_M^A\right)}{T_M^A} + C\frac{L_B \left(T - T_M^B\right)}{T_M^B}\right\} p'(\phi) + \left\{(1 - C)\frac{W_A}{2} + C\frac{W_B}{2}\right\} g'(\phi) \qquad (9.5)$$

and

$$\frac{\partial f}{\partial C} = \left[f_B(\phi, T) + \frac{RT}{V_m} \ln C\right] - \left[f_A(\phi, T) + \frac{RT}{V_m} \ln(1 - C)\right]$$

$$= \left[\mu_B(\phi, C, T) - \mu_A(\phi, C, T)\right] / V_m \qquad (9.6)$$

where μ_A and μ_B are the classical chemical potentials for the binary species. $p'(\phi)$ and $g'(\phi)$ are the partial derivatives of of p and g with respect to ϕ

```
>>> def pPrime(phi):
...     return 30. * g(phi)

>>> def gPrime(phi):
...     return 2. * phi * (1 - phi) * (1 - 2 * phi)
```

V_m is the molar volume, which we take to be independent of concentration and phase

```
>>> Vm = 7.42 # cm**3 / mol
```

On comparison with `examples.phase.simple.input`, we can see that the present form of the phase field equation is identical to the one found earlier, with the source now composed of the concentration-weighted average of the source for either pure component. We let the pure component barriers equal the previous value

```
>>> deltaA = deltaB = 1.5 * dx
>>> sigmaA = 3.7e-5 # J / cm**2
>>> sigmaB = 2.9e-5 # J / cm**2
>>> betaA = 0.33 # cm / (K s)
>>> betaB = 0.39 # cm / (K s)
>>> kappaA = 6 * sigmaA * deltaA # J / cm
>>> kappaB = 6 * sigmaB * deltaB # J / cm
>>> WA = 6 * sigmaA / deltaA # J / cm**3
>>> WB = 6 * sigmaB / deltaB # J / cm**3
```

and define the averages

```
>>> W = (1 - C) * WA / 2. + C * WB / 2.
>>> enthalpy = (1 - C) * enthalpyA + C * enthalpyB
```

We can now linearize the source exactly as before

```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> dmPhidPhi = 2 * W - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase * (S1 < 0)
```

Using the same gradient energy coefficient and phase field mobility

```
>>> kappa = (1 - C) * kappaA + C * kappaB
>>> Mphi = TmA * betaA / (6 * LA * deltaA)
```

we define the phase field equation

```
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> from fipy.terms.implicitSourceTerm import ImplicitSourceTerm


>>> phaseEq = TransientTerm(1/Mphi) == ImplicitDiffusionTerm(coeff=kappa) \
...    + S0 + ImplicitSourceTerm(coeff=S1 * (S1 < 0))
```

When coding explicitly, it is typical to simply write a function to evaluate the chemical potentials μ_A and μ_B and then perform the finite differences necessary to calculate their gradient and divergence, e.g.,

```
def deltaChemPot(phase, C, T):
    return ((Vm * (enthalpyB * p(phase) + WA * g(phase)) + R * T * log(1 - C)) -
            (Vm * (enthalpyA * p(phase) + WA * g(phase)) + R * T * log(C)))

for j in range(faces):
    flux[j] = ((Mc[j+.5] + Mc[j-.5]) / 2) \
        * (deltaChemPot(phase[j+.5], C[j+.5], T) \
          - deltaChemPot(phase[j-.5], C[j-.5], T)) / dx

for j in range(cells):
    diffusion = (flux[j+.5] - flux[j-.5]) / dx
```

where we neglect the details of the outer boundaries (j = 0 and j = N) or exactly how to translate j+.5 or j-.5 into an array index, much less the complexities of higher dimensions. FiPy can handle all of these issues automatically, so we could just write:

```
chemPotA = Vm * (enthalpyA * p(phase) + WA * g(phase)) + R * T * log(C)
chemPotB = Vm * (enthalpyB * p(phase) + WB * g(phase)) + R * T * log(1-C)
flux = Mc * (chemPotB - chemPotA).getFaceGrad()
eq = TransientTerm() == flux.getDivergence()
```

Although the second syntax would essentially work as written, such an explicit implementation would be very slow. In order to take advantage of FiPy's implicit solvers, it is necessary to reduce Eq. (9.4) to the canonical form of Eq. (3.2), hence we must expand Eq. (9.6) as

$$\frac{\partial f}{\partial C} = \left[\frac{L_B \left(T - T_M^B \right)}{T_M^B} - \frac{L_A \left(T - T_M^A \right)}{T_M^A} \right] p(\phi) + \frac{RT}{V_m} \left[\ln C - \ln(1 - C) \right] + \frac{W_B - W_A}{2} g(\phi)$$

In either bulk phase, $\nabla p(\phi) = \nabla g(\phi) = 0$, so we can then reduce Eq. (9.4) to

$$\frac{\partial C}{\partial t} = \nabla \cdot \left(M_C \nabla \left\{ \frac{RT}{V_m} \left[\ln C - \ln(1 - C) \right] \right\} \right)$$

$$= \nabla \cdot \left[\frac{M_C RT}{C(1 - C)V_m} \nabla C \right] \tag{9.7}$$

and, by comparison with Fick's second law

$$\frac{\partial C}{\partial t} = \nabla \cdot \left[D \nabla C \right],$$

we can associate the mobility M_C with the intrinsic diffusivity D by $M_C \equiv DC(1 - C)V_m/RT$ and write Eq. (9.4) as

$$\frac{\partial C}{\partial t} = \nabla \cdot (D \nabla C)$$

$$+ \nabla \cdot \left(\frac{DC(1 - C)V_m}{RT} \left\{ \left[\frac{L_B \left(T - T_M^B \right)}{T_M^B} - \frac{L_A \left(T - T_M^A \right)}{T_M^A} \right] \nabla p(\phi) + \frac{W_B - W_A}{2} \nabla g(\phi) \right\} \right). \tag{9.8}$$

The first term is clearly a `DiffusionTerm`. The second is less obvious, but by factoring out C, we can see that this is a `ConvectionTerm` with a velocity

$$\vec{u}_\phi = \frac{D(1 - C)V_m}{RT} \left\{ \left[\frac{L_B \left(T - T_M^B \right)}{T_M^B} - \frac{L_A \left(T - T_M^A \right)}{T_M^A} \right] \nabla p(\phi) + \frac{W_B - W_A}{2} \nabla g(\phi) \right\}$$

due to phase transformation, such that

$$\frac{\partial C}{\partial t} = \nabla \cdot (D \nabla C) + \nabla \cdot (C \vec{u}_\phi)$$

or

```
>>> Dl = Variable(value=1e-5) # cm**2 / s
>>> Ds = Variable(value=1e-9) # cm**2 / s
>>> D = (Dl - Ds) * phase.getArithmeticFaceValue() + Dl
>>> diffusion = ImplicitDiffusionTerm(coeff=D)

>>> phaseTransformationVelocity = \
...   ((enthalpyB - enthalpyA) * p(phase).getFaceGrad()
...    + 0.5 * (WB - WA) * g(phase).getFaceGrad()) \
...    * D * (1. - C).getHarmonicFaceValue() * Vm / (R * T)
```

```
>>> from fipy.terms.powerLawConvectionTerm import PowerLawConvectionTerm
>>> diffusionEq = TransientTerm() == diffusion \
...     + PowerLawConvectionTerm(coeff=phaseTransformationVelocity,
...                              diffusionTerm=diffusion)
```

We initialize the phase field to a step function in the middle of the domain

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=mesh.getCellCenters()[...,0] > L/2.)
```

and start with a uniform composition field $C = 1/2$

```
>>> C.setValue(0.5)
```

In equilibrium, $\mu_A(0, C_L, T) = \mu_A(1, C_S, T)$ and $\mu_B(0, C_L, T) = \mu_B(1, C_S, T)$ and, for ideal solutions, we can deduce the liquidus and solidus compositions as

$$C_L = \frac{1 - \exp\left(-\frac{L_A\left(T - T_M^A\right)}{T_M^A}\frac{V_m}{RT}\right)}{\exp\left(-\frac{L_B\left(T - T_M^B\right)}{T_M^B}\frac{V_m}{RT}\right) - \exp\left(-\frac{L_A\left(T - T_M^A\right)}{T_M^A}\frac{V_m}{RT}\right)}$$

$$C_S = \exp\left(-\frac{L_B\left(T - T_M^B\right)}{T_M^B}\frac{V_m}{RT}\right)C_L$$

```
>>> from fipy.tools.numerix import exp
>>> Cl = (1. - exp(-enthalpyA * Vm / (R * T))) \
...    / (exp(-enthalpyB * Vm / (R * T)) - exp(-enthalpyA * Vm / (R * T)))
>>> Cs = exp(-enthalpyB * Vm / (R * T)) * Cl
```

The phase fraction is predicted by the lever rule

```
>>> Cavg = C.getCellVolumeAverage()
>>> fraction = (Cl - Cavg) / (Cl - Cs)
```

For the special case of `fraction = Cavg = 0.5`, a little bit of algebra reveals that the temperature that leaves the phase fraction unchanged is given by

```
>>> T.setValue((LA + LB) * TmA * TmB / (LA * TmB + LB * TmA))
```

In this simple, binary, ideal solution case, we can derive explicit expressions for the solidus and liquidus compositions. In general, this may not be possible or practical. In that event, the root-finding facilities in SciPy can be used. We'll need a function to return the two conditions for equilibrium

$$0 = \mu_A(1, C_S, T) - \mu_A(0, C_L, T) = \frac{L_A\left(T - T_M^A\right)}{T_M^A}V_m + RT\ln(1 - C_S) - RT\ln(1 - C_L)$$

$$0 = \mu_B(1, C_S, T) - \mu_B(0, C_L, T) = \frac{L_B\left(T - T_M^B\right)}{T_M^B}V_m + RT\ln C_S - RT\ln C_L$$

```
>>> from fipy.tools.numerix import log
>>> def equilibrium(C):
...     return [enthalpyA * Vm + R * T * log(1 - C[0]) - R * T * log(1 - C[1]),
...             enthalpyB * Vm + R * T * log(C[0]) - R * T * log(C[1])]
```

and we'll have much better luck if we also supply the Jacobian

$$
\begin{bmatrix}
\frac{\partial(\mu_A^S - \mu_A^L)}{\partial C_S} & \frac{\partial(\mu_A^S - \mu_A^L)}{\partial C_L} \\
\frac{\partial(\mu_B^S - \mu_B^L)}{\partial C_S} & \frac{\partial(\mu_B^S - \mu_B^L)}{\partial C_L}
\end{bmatrix}
= RT
\begin{bmatrix}
-\frac{1}{1-C_S} & \frac{1}{1-C_L} \\
\frac{1}{C_S} & -\frac{1}{C_L}
\end{bmatrix}
$$

```
>>> from fipy.tools.numerix import array
>>> def equilibriumJacobian(C):
...     return R * T * array([[-1. / (1 - C[0]), 1. / (1 - C[1])],
...                           [ 1. / C[0],       -1. / C[1]]])
```

```
>>> try:
...     from scipy.optimize import fsolve
...     CsRoot, ClRoot = fsolve(func=equilibrium, x0=[0.5, 0.5],
...                             fprime=equilibriumJacobian)
... except ImportError:
...     ClRoot = CsRoot = 0
...     print "The SciPy library is not available to calculate the solidus and \
... liquidus concentrations"
```

```
>>> print Cl.allclose(ClRoot)
1
>>> print Cs.allclose(CsRoot)
1
```

We plot the result against the sharp interface solution

```
>>> sharp = CellVariable(name="sharp", mesh=mesh)
>>> x = mesh.getCellCenters()[...,0]
>>> sharp.setValue(Cs, where=x < L * fraction)
>>> sharp.setValue(Cl, where=x >= L * fraction)
```

```
>>> if __name__ == '__main__':
...     from fipy import viewers
...     viewer = viewers.make(vars=(phase, C, sharp),
...                           limits={'datamin': 0., 'datamax': 1.})
...     viewer.plot()
```

Because the phase field interface will not move, and because we've seen in earlier examples that the diffusion problem is unconditionally stable, we need take only one very large timestep to reach equilibrium

```
>>> dt = 1.e2
```

Because the phase field equation is coupled to the composition through `enthalpy` and `W` and the diffusion equation is coupled to the phase field through `phaseTransformationVelocity`, it is necessary sweep this non-linear problem to convergence. We use the "residual" of the equations (a measure of how well they think they have solved the given set of linear equations) as a test for how long to sweep. Because of the `ConvectionTerm`, the solution matrix for `diffusionEq` is asymmetric and cannot be solved by the default `LinearPCGSolver`. Therefore, we use a `LinearLUSolver` for this equation.
We now use the "`sweep()`" method instead of "`solve()`" because we require the residual.

```
>>> from fipy.solvers.linearLUSolver import LinearLUSolver
>>> solver = LinearLUSolver(tolerance=1e-10)

>>> phase.updateOld()
>>> C.updateOld()
>>> phaseRes = 1e+10
>>> diffRes = 1e+10
>>> while phaseRes > 1e-3 or diffRes > 1e-3:
...     phaseRes = phaseEq.sweep(var=phase, dt=dt)
...     diffRes = diffusionEq.sweep(var=C, dt=dt, solver=solver)
>>> if __name__ == '__main__':
...     viewer.plot()
...     raw_input("stationary phase field")
```
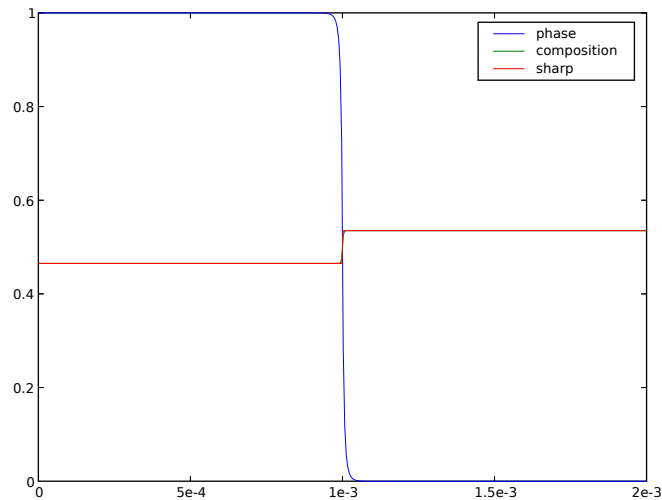

We verify that the bulk phases have shifted to the predicted solidus and liquidus compositions

```
>>> print Cs.allclose(C[0], atol=2e-4)
1
>>> print Cl.allclose(C[nx-1], atol=2e-4)
1
```

and that the phase fraction remains unchanged

```
>>> print fraction.allclose(phase.getCellVolumeAverage(), atol=2e-4)
1
```

while conserving mass overall

```
>>> print Cavg.allclose(0.5, atol=1e-8)
1
```

We now quench by ten degrees

```
>>> T.setValue(T() - 10.) # K

>>> sharp.setValue(Cs, where=x < L * fraction)
>>> sharp.setValue(Cl, where=x >= L * fraction)
```

Because this lower temperature will induce the phase interface to move (solidify), we will need to take much smaller timesteps (the time scales of diffusion and of phase transformation compete with each other).

```
>>> dt = 1.e-6

>>> for i in range(100):
...     phase.updateOld()
...     C.updateOld()
...     phaseRes = 1e+10
...     diffRes = 1e+10
...     while phaseRes > 1e-3 or diffRes > 1e-3:
...         phaseRes = phaseEq.sweep(var=phase, dt=dt)
...         diffRes = diffusionEq.sweep(var=C, dt=dt, solver=solver)
...     if __name__ == '__main__':
...         viewer.plot()

>>> if __name__ == '__main__':
...     raw_input("moving phase field")
```
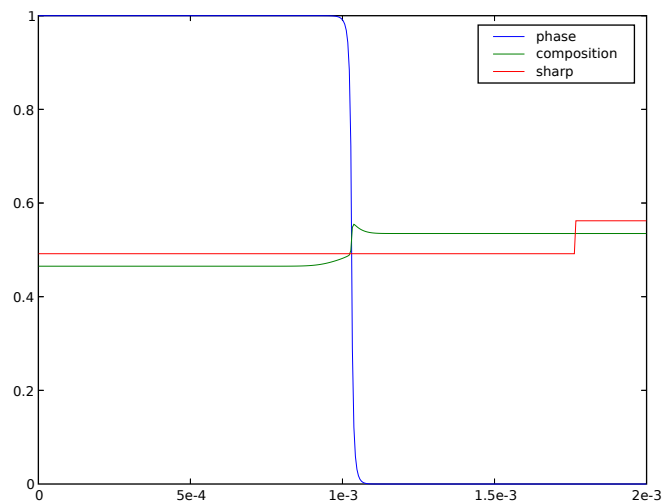

We see that the composition on either side of the interface approach the sharp-interface solidus and liquidus, but it will take a great many more timesteps to reach equilibrium. If we waited sufficiently long, we could again verify the final concentrations and phase fraction against the expected values.

9.3   Module `examples.phase.quaternary`

   The same procedure used to construct the two-component phase field diffusion problem in `examples.phase.binary` can be used to build up a system of multiple components. Once again, we'll focus on 1D.

```
>>> nx = 400
>>> dx = 0.01
>>> L = nx * dx
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = dx, nx = nx)
```

We consider a free energy density $f(\phi, C_0, \ldots, C_N, T)$ that is a function of phase ϕ

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phase = CellVariable(mesh=mesh, name='phase', value=1, hasOld=1)
```

interstitial components $C_0 \ldots C_M$

```
>>> interstitials = [
...     CellVariable(mesh=mesh, name='C0', hasOld=1)
... ]
```

substitutional components $C_{M+1} \ldots C_{N-1}$

```
>>> substitutionals = [
...     CellVariable(mesh=mesh, name='C1', hasOld=1),
...     CellVariable(mesh=mesh, name='C2', hasOld=1),
... ]
```

a "solvent" C_N that is constrained by the concentrations of the other substitutional species, such that $C_N = 1 - \sum_{j=M}^{N-1} C_j$,

```
>>> solvent = 1
>>> for Cj in substitutionals:
...     solvent -= Cj
>>> solvent.name = 'CN'
```

and temperature T

```
>>> T = 1000
```

The free energy density of such a system can be written as

$$f(\phi, C_0, \ldots, C_N, T) = \sum_{j=0}^{N} C_j \left[\mu_j^\circ(\phi, T) + RT \ln \frac{C_j}{\rho} \right]$$

where

```
>>> R = 8.314 # J / (mol K)
```

is the gas constant. As in the binary case,

$$\mu_j^\circ(\phi, T) = p(\phi)\mu_j^{\circ S}(T) + (1 - p(\phi)) \, \mu_j^{\circ L}(T) + \frac{W_j}{2} g(\phi)$$

is constructed with the free energies of the pure components in each phase, given the "tilting" function

```
>>> def p(phi):
...     return phi**3 * (6 * phi**2 - 15 * phi + 10)
```

and the "double well" function

```
>>> def g(phi):
...     return (phi * (1 - phi))**2
```

We consider a very simplified model that has partial molar volumes $\bar{V}_0 = \cdots = \bar{V}_M = 0$ for the "interstitials" and $\bar{V}_{M+1} = \cdots = \bar{V}_N = 1$ for the "substitutionals". This approximation has been used in a number of models where density effects are ignored, including the treatment of electrons in electrodeposition processes [30, 31]. Under these constraints

$$\frac{\partial f}{\partial \phi} = \sum_{j=0}^{N} C_j \frac{\partial f_j}{\partial \phi}$$

$$= \sum_{j=0}^{N} C_j \left[\mu_j^{\circ SL}(T) p'(\phi) + \frac{W_j}{2} g'(\phi) \right]$$

$$\frac{\partial f}{\partial C_j} = \left[\mu_j^{\circ}(\phi, T) + RT \ln \frac{C_j}{\rho} \right]$$

$$= \mu_j(\phi, C_j, T) \qquad \text{for } j = 0 \ldots M$$

and

$$\frac{\partial f}{\partial C_j} = \left[\mu_j^{\circ}(\phi, T) + RT \ln \frac{C_j}{\rho} \right] - \left[\mu_N^{\circ}(\phi, T) + RT \ln \frac{C_N}{\rho} \right]$$

$$= [\mu_j(\phi, C_j, T) - \mu_N(\phi, C_N, T)] \qquad \text{for } j = M+1 \ldots N-1$$

where $\mu_j^{\circ SL}(T) \equiv \mu_j^{\circ S}(T) - \mu_j^{\circ L}(T)$ and where μ_j is the classical chemical potential of component j for the binary species and $\rho = 1 + \sum_{j=0}^{M} C_j$ is the total molar density.

```
>>> rho = 1.
>>> for Cj in interstitials:
...     rho += Cj
```

$p'(\phi)$ and $g'(\phi)$ are the partial derivatives of of p and g with respect to ϕ

```
>>> def pPrime(phi):
...     return 30. * g(phi)
```

```
>>> def gPrime(phi):
...     return 2. * phi * (1 - phi) * (1 - 2 * phi)
```

We "cook" the standard potentials to give the desired solid and liquid concentrations, with a solid phase rich in interstitials and the solvent and a liquid phase rich in the two substitutional species.

```
>>> interstitials[0].S = 0.3
>>> interstitials[0].L = 0.4
>>> substitutionals[0].S = 0.4
>>> substitutionals[0].L = 0.3
```

```
>>> substitutionals[1].S = 0.2
>>> substitutionals[1].L = 0.1
>>> solvent.S = 1.
>>> solvent.L = 1.
>>> for Cj in substitutionals:
...     solvent.S -= Cj.S
...     solvent.L -= Cj.L


>>> rhoS = rhoL = 1.
>>> for Cj in interstitials:
...     rhoS += Cj.S
...     rhoL += Cj.L



>>> from fipy.tools.numerix import log
>>> for Cj in interstitials + substitutionals + [solvent]:
...     Cj.standardPotential = R * T * (log(Cj.L/rhoL) - log(Cj.S/rhoS))

>>> for Cj in interstitials:
...     Cj.diffusivity = 1.
...     Cj.barrier = 0.

>>> for Cj in substitutionals:
...     Cj.diffusivity = 1.
...     Cj.barrier = R * T

>>> solvent.barrier = R * T
```

We create the phase equation

$$\frac{1}{M_\phi}\frac{\partial\phi}{\partial t} = \kappa_\phi\nabla^2\phi - \sum_{j=0}^{N} C_j\left[\mu_j^{\circ SL}(T)p'(\phi) + \frac{W_j}{2}g'(\phi)\right]$$

with a semi-implicit source just as in `examples.phase.simple.input` and `examples.phase.binary`

```
>>> enthalpy = 0.
>>> barrier = 0.
>>> for Cj in interstitials + substitutionals + [solvent]:
...     enthalpy += Cj * Cj.standardPotential
...     barrier += Cj * Cj.barrier

>>> mPhi = -((1 - 2 * phase) * barrier + 30 * phase * (1 - phase) * enthalpy)
>>> dmPhidPhi = 2 * barrier - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase * (S1 < 0)
```

```
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> from fipy.terms.implicitSourceTerm import ImplicitSourceTerm

>>> phase.mobility = 1.
>>> phase.gradientEnergy = 25
>>> phase.equation = TransientTerm(coeff=1/phase.mobility) \
...     == ImplicitDiffusionTerm(coeff=phase.gradientEnergy) \
...         + S0 + ImplicitSourceTerm(coeff = S1 * (S1 < 0))
```

We could construct the diffusion equations one-by-one, in the manner of `examples.phase.binary`, but it is better to take advantage of the full scripting power of the Python language, where we can easily loop over components or even make "factory" functions if we desire. For the interstitial diffusion equations, we arrange in canonical form as before:

$$
\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = \underbrace{D_j \nabla^2 C_j}_{\text{diffusion}}
$$

$$
+ \underbrace{D_j \nabla \cdot \frac{C_j}{1 + \sum_{\substack{k=0 \\ k \neq j}}^{M} C_k} \left\{ \overbrace{\frac{\rho}{RT} \left[\mu_j^{\circ SL} \nabla p(\phi) + \frac{W_j}{2} \nabla g(\phi) \right]}^{\text{phase transformation}} - \overbrace{\sum_{\substack{i=0 \\ i \neq j}}^{M} \nabla C_i}^{\text{counter diffusion}} \right\}}_{\text{convection}}
$$

```
>>> from fipy.terms.powerLawConvectionTerm import PowerLawConvectionTerm

>>> for Cj in interstitials:
...     phaseTransformation = (rho.getHarmonicFaceValue() / (R * T)) \
...       * (Cj.standardPotential * p(phase).getFaceGrad()
...           + 0.5 * Cj.barrier * g(phase).getFaceGrad())
...
...     CkSum = CellVariable(mesh=mesh, value=0.)
...     for Ck in [Ck for Ck in interstitials if Ck is not Cj]:
...         CkSum += Ck
...
...     counterDiffusion = CkSum.getFaceGrad()
...
...     convectionCoeff = counterDiffusion + phaseTransformation
...     convectionCoeff *= (Cj.diffusivity
...                         / (1. + CkSum.getHarmonicFaceValue()))
...
...     diffusionTerm = ImplicitDiffusionTerm(coeff=Cj.diffusivity)
...     convectionTerm = PowerLawConvectionTerm(coeff=convectionCoeff,
...                                             diffusionTerm=diffusionTerm)
...
...     Cj.equation = TransientTerm() == diffusionTerm + convectionTerm
```

The canonical form of the substitutional diffusion equations is

$$\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = \underbrace{D_j \nabla^2 C_j}_{\text{diffusion}}$$

$$+ \underbrace{D_j \nabla \cdot \frac{C_j}{1 - \sum_{\substack{k=2 \\ k \neq j}}^{n-1} C_k} \left\{ \overbrace{\frac{C_N}{RT} \left[\left(\mu_j^{\circ SL} - \mu_N^{\circ SL} \right) \nabla p(\phi) + \frac{W_j - W_N}{2} \nabla g(\phi) \right]}^{\text{phase transformation}} + \overbrace{\sum_{\substack{i=M+1 \\ i \neq j}}^{N} \nabla C_i}^{\text{counter diffusion}} \right\}}_{\text{convection}}$$

```
>>> for Cj in substitutionals:
...     phaseTransformation = (solvent.getHarmonicFaceValue() / (R * T)) \
...         * ((Cj.standardPotential - solvent.standardPotential) * p(phase).getFaceGrad()
...            + 0.5 * (Cj.barrier - solvent.barrier) * g(phase).getFaceGrad())
...
...     CkSum = CellVariable(mesh=mesh, value=0.)
...     for Ck in [Ck for Ck in substitutionals if Ck is not Cj]:
...         CkSum += Ck
...
...     counterDiffusion = CkSum.getFaceGrad()
...
...     convectionCoeff = counterDiffusion + phaseTransformation
...     convectionCoeff *= (Cj.diffusivity
...                         / (1. - CkSum.getHarmonicFaceValue()))
...
...     diffusionTerm = ImplicitDiffusionTerm(coeff=Cj.diffusivity)
...     convectionTerm = PowerLawConvectionTerm(coeff=convectionCoeff,
...                                             diffusionTerm=diffusionTerm)
...
...     Cj.equation = TransientTerm() == diffusionTerm + convectionTerm
```

We start with a sharp phase boundary

$$\xi = \begin{cases} 1 & \text{for } x \leq L/2, \\ 0 & \text{for } x > L/2, \end{cases}$$

```
>>> x = mesh.getCellCenters()[...,0]
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L / 2)
```

and with uniform concentration fields, initially equal to the average of the solidus and liquidus concentrations

```
>>> for Cj in interstitials + substitutionals:
...     Cj.setValue((Cj.S + Cj.L) / 2.)
```

If we're running interactively, we create a viewer

```
>>> if __name__ == '__main__':
...     from fipy import viewers
...     viewer = viewers.make(vars = [phase] \
...                                   + interstitials + substitutionals \
...                                   + [solvent],
...                           limits = {'datamin': 0, 'datamax': 1})
...     viewer.plot()
```

and again iterate to equilibrium

```
>>> from fipy.solvers.linearLUSolver import LinearLUSolver
>>> solver = LinearLUSolver(tolerance=1e-3)

>>> dt = 10000
>>> for i in range(5):
...     for field in [phase] + substitutionals + interstitials:
...         field.updateOld()
...     phase.equation.solve(var = phase, dt = dt)
...     for field in substitutionals + interstitials:
...         field.equation.solve(var = field,
...                              dt = dt,
...                              solver = solver)
...     if __name__ == '__main__':
...         viewer.plot()
```
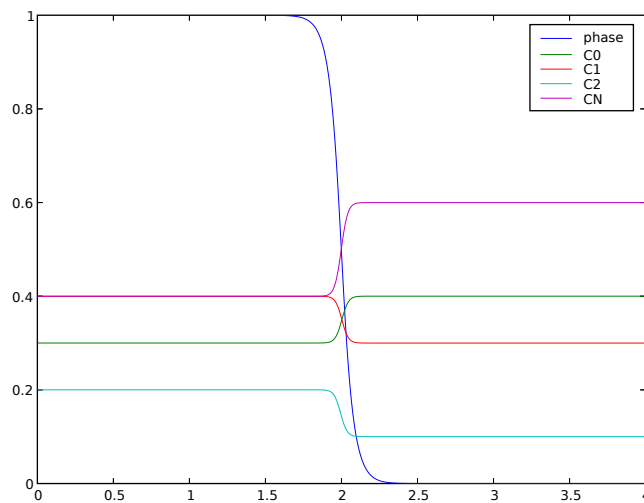

We can confirm that the far-field phases have remained separated

```
>>> from fipy.tools.numerix import take, allclose
>>> ends = take(phase, (0,-1))
>>> allclose(ends, (1.0, 0.0), rtol = 1e-5, atol = 1e-5)
1
```

and that the concentration fields have appropriately segregated into their equilibrium values in each phase

```
>>> equilibrium = True
>>> for Cj in interstitials + substitutionals:
...       ends = take(Cj, (0,-1))
...       equilibrium &= allclose(ends, (Cj.S, Cj.L), rtol = 3e-3, atol = 3e-3)
>>> print equilibrium
1
```

9.4   Module `examples.phase.anisotropy.input`

In this example we solve a coupled phase and temperature equation to model solidification, and eventually dendritic growth, based on the work of Warren, Kobayashi, Lobkovsky and Carter [26]
We start from a circular seed in a 2D mesh:

```
>>> numberOfCells = 40
>>> Length = numberOfCells * 2.5 / 100.
>>> nx = numberOfCells
>>> ny = numberOfCells
>>> dx = Length / nx
>>> dy = Length / ny
>>> radius = Length / 4.
>>> seedCenter = (Length / 2., Length / 2.)
>>> initialTemperature = -0.4
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

Dendritic growth will not be observed with this small test system. If you wish to see dendritic growth reset the following parameters such that `numberOfCells = 500`, `steps = 10000`, `radius = dx * 5.` `seedCenter = (0. , 0.)` and `initialTemperature = -0.5`.
The governing equation for the phase field is given by:

$$\tau_\phi \frac{\partial \phi}{\partial t} = \nabla \cdot [D\nabla\phi + A\nabla\xi] + \phi(1-\phi)m(\phi, T)$$

where

$$m(\phi, T) = \phi - \frac{1}{2} - \frac{\kappa_1}{\pi}\arctan\left(\kappa_2 T\right).$$

The coefficients D and A are given by,

$$D = \alpha^2 \left[1 + c\beta\right]^2$$

and

$$A = \alpha^2 c \left[1 + c\beta\right]\beta_\psi$$

where $\beta = \frac{1-\Phi^2}{1+\Phi^2}$, $\Phi = \tan\left(\frac{N}{2}\psi\right)$, $\psi = \theta + \arctan\frac{\phi_y}{\phi_x}$ and $\xi_x = -\phi_y$ and $\xi_y = \phi_x$.
The governing equation for temperature is given by:

$$\frac{\partial T}{\partial t} = D_T \nabla^2 T + \frac{\partial \phi}{\partial t}$$

Here the phase and temperature equations are solved with an explicit and implicit technique, respectively.
The parameters for these equations are

```
>>> timeStepDuration = 5e-5
>>> tau = 3e-4
>>> alpha = 0.015
>>> c = 0.02
>>> N = 4.
>>> kappa1 = 0.9
>>> kappa2 = 20.
>>> tempDiffusionCoeff = 2.25
>>> theta = 0.
```

The `phase` variable is 0 for a liquid and 1 for a solid. Here, the `phase` variable is initialized as a liquid,

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phase = CellVariable(name='phase field', mesh=mesh, hasOld=1)
```

The `hasOld` flag keeps the old value of the variable. This is necessary for a transient solution. In this example we wish to set up an interior region that is solid. A value of 1 is assigned to the `phase` variable on a patch defined by the method:
The domain is seeded with a circular solidified region with parameters `seedCenter` and `radius` representing the center and radius of the seed.

```
>>> x, y = mesh.getCellCenters()[...,0], mesh.getCellCenters()[...,1]
>>> phase.setValue(1., where=((x - seedCenter[0])**2
...                           + (y - seedCenter[1])**2) < radius**2)
```

The temperature field is initialized to a value of -0.4 throughout:

```
>>> temperature = CellVariable(
...     name='temperature',
...     mesh=mesh,
...     value=initialTemperature,
...     hasOld=1
...     )
```

The $m(\phi, T)$ variable is created from the `phase` and `temperature` variables.

```
>>> from fipy.tools.numerix import pi, arctan, arctan2, tan
>>> mVar = phase - 0.5 - kappa1 / pi * arctan(kappa2 * temperature)
```

The following section of code builds up the A and D coefficients.

```
>>> phaseY = phase.getFaceGrad().dot((0, 1))
>>> phaseX = phase.getFaceGrad().dot((1, 0))
>>> psi = theta + arctan2(phaseY, phaseX)
>>> Phi = tan(N * psi / 2)
>>> PhiSq = Phi**2
>>> beta = (1. - PhiSq) / (1. + PhiSq)
>>> betaPsi = -N * 2 * Phi / (1 + PhiSq)
>>> A = alpha**2 * c * (1.+ c * beta) * betaPsi
>>> D = alpha**2 * (1.+ c * beta)**2
```

The $\nabla\xi$ variable (`dxi`), given by $(\xi_x, \xi_y) = (-\phi_y, \phi_x)$, is constructed by first obtaining $\nabla\phi$ using `getFaceGrad()`. The axes are then swapped using `_take((1,0))` and finally the vector is multiplied by `(-1, 1)` to negate the x component.

```
>>> dxi = phase.getFaceGrad()._take((1, 0), axis = 1) * (-1, 1)
>>> anisotropySource = (A * dxi).getDivergence()
```

The phase equation can now be constructed.

```
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.explicitDiffusionTerm import ExplicitDiffusionTerm
>>> from fipy.terms.implicitSourceTerm import ImplicitSourceTerm
>>> phaseEq = TransientTerm(tau) == ExplicitDiffusionTerm(D) + \
...     ImplicitSourceTerm(mVar * ((mVar < 0) - phase)) + \
...     ((mVar > 0.) * mVar * phase + anisotropySource)
```

The temperature equation is built in the following way,

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> temperatureEq = TransientTerm() == \
...                 ImplicitDiffusionTerm(tempDiffusionCoeff) + \
...                 (phase - phase.getOld()) / timeStepDuration
```

If we are running this example interactively, we create viewers for the phase and temperature fields

```
>>> if __name__ == '__main__':
...     from fipy.viewers import make
...     phaseViewer = make(vars=phase)
...     temperatureViewer = make(vars=temperature,
...                              limits={'datamin': -0.5, 'datamax': 0.5})
...     phaseViewer.plot()
...     temperatureViewer.plot()
```

we iterate the solution in time, plotting as we go if running interactively,

```
>>> steps = 10
>>> for i in range(steps):
...     phase.updateOld()
...     temperature.updateOld()
...     phaseEq.solve(phase, dt=timeStepDuration)
...     temperatureEq.solve(temperature, dt=timeStepDuration)
...     if i%10 == 0 and __name__ == '__main__':
...         phaseViewer.plot()
...         temperatureViewer.plot()
```

The solution is compared with test data. The test data was created for `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `test.gz` extracts the data and compares it with the `phase` variable.

```
>>> import examples.phase.anisotropy
>>> import os
>>> filepath = os.path.join(examples.phase.anisotropy.__path__[0], 'test.gz')
>>> from fipy.tools import dump
>>> testData = dump.read(filepath)
>>> from fipy.tools.numerix import allclose
>>> print allclose(phase, testData)
1
```

9.5 Module `examples.phase.impingement.mesh40x1.input`

In this example we solve a coupled phase and orientation equation on a one dimensional grid. This is another aspect of the model of Warren, Kobayashi, Lobkovsky and Carter [26]

```
>>> nx = 40
>>> Lx = 2.5 * nx / 100.
>>> dx = Lx / nx
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx=dx, nx=nx)
```

This problem simulates the wet boundary that forms between grains of different orientations. The phase equation is given by

$$\tau_\phi \frac{\partial \phi}{\partial t} = \alpha^2 \nabla^2 \phi + \phi(1 - \phi)m_1(\phi, T) - 2s\phi|\nabla\theta| - \epsilon^2\phi|\nabla\theta|^2$$

where

$$m_1(\phi, T) = \phi - \frac{1}{2} - T\phi(1 - \phi)$$

and the orientation equation is given by

$$P(\epsilon|\nabla\theta|)\tau_\theta\phi^2 \frac{\partial \theta}{\partial t} = \nabla \cdot \left[\phi^2\left(\frac{s}{|\nabla\theta|} + \epsilon^2\right)\nabla\theta\right]$$

where

$$P(w) = 1 - \exp(-\beta w) + \frac{\mu}{\epsilon}\exp(-\beta w)$$

The initial conditions for this problem are set such that $\phi = 1$ for $0 \leq x \leq L_x$ and

$$\theta = \begin{cases} 1 & \text{for } 0 \leq x < L_x/2, \\ 0 & \text{for } L_x/2 \leq x \leq L_x. \end{cases}$$

Here the phase and orientation equations are solved with an explicit and implicit technique respectively. The parameters for these equations are

```
>>> timeStepDuration = 0.02
>>> phaseTransientCoeff = 0.1
>>> thetaSmallValue = 1e-6
>>> beta = 1e5
>>> mu = 1e3
>>> thetaTransientCoeff = 0.01
>>> gamma= 1e3
>>> epsilon = 0.008
>>> s = 0.01
>>> alpha = 0.015
```

The system is held isothermal at

```
>>> temperature = 1.
```

and is initially solid everywhere

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phase = CellVariable(
...     name='phase field',
...     mesh=mesh,
...     value=1.
...     )
```

Because `theta` is an S^1-valued variable (i.e. it maps to the circle) and thus intrinsically has 2π-peridocity, we must use `ModularVariable` instead of a `CellVariable`. A `ModularVariable` confines `theta` to $-\pi < \theta \leq \pi$ by adding or subtracting 2π where necessary and by defining a new subtraction operator between two angles.

```
>>> from fipy.variables.modularVariable import ModularVariable
>>> theta = ModularVariable(
...     name='theta',
...     mesh=mesh,
...     value=1.,
...     hasOld=1
...     )
```

The left and right halves of the domain are given different orientations.

```
>>> theta.setValue(0., where=mesh.getCellCenters()[...,0] > Lx / 2.)
```

The `phase` equation is built in the following way.

```
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.explicitDiffusionTerm import ExplicitDiffusionTerm
>>> from fipy.terms.implicitSourceTerm import ImplicitSourceTerm

>>> mPhiVar = phase - 0.5 + temperature * phase * (1 - phase)
```

The source term is linearized in the manner demonstrated in `examples.phase.simple.input` (Kobayashi, semi-implicit).

```
>>> thetaMag = theta.getOld().getGrad().getMag()
>>> implicitSource = mPhiVar * (phase - (mPhiVar < 0))
>>> implicitSource += (2 * s + epsilon**2 * thetaMag) * thetaMag
```

The `phase` equation is constructed.

```
>>> phaseEq = TransientTerm(phaseTransientCoeff) \
...    == ExplicitDiffusionTerm(alpha**2) \
...       - ImplicitSourceTerm(implicitSource) \
...       + (mPhiVar > 0) * mPhiVar * phase
```

The `theta` equation is built in the following way. The details for this equation are fairly involved, see J.A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of `theta` on the circle.

```
>>> phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
>>> phaseModSq = phaseMod * phaseMod
>>> expo = epsilon * beta * theta.getGrad().getMag()
>>> expo = (expo < 100.) * (expo - 100.) + 100.
>>> from fipy.tools.numerix import exp
>>> pFunc = 1. + exp(-expo) * (mu / epsilon - 1.)
```

```
>>> phaseFace = phase.getArithmeticFaceValue()
>>> phaseSq = phaseFace * phaseFace
>>> gradMag = theta.getFaceGrad().getMag()
>>> eps = 1. / gamma / 10.
>>> gradMag += (gradMag < eps) * eps
>>> IGamma = (gradMag > 1. / gamma) * (1 / gradMag - gamma) + gamma
>>> diffusionCoeff = phaseSq * (s * IGamma + epsilon**2)
```

The source term requires the evaluation of the face gradient without the modular operator. A method of `ModularVariable`, `getFaceGradNoMod()`, eveluates the gradient without modular arithmetic.

```
>>> thetaGradDiff = theta.getFaceGrad() - theta.getFaceGradNoMod()
>>> sourceCoeff = (diffusionCoeff * thetaGradDiff).getDivergence()
```

Finally the `theta` equation can be constructed.

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> thetaEq = TransientTerm(thetaTransientCoeff * phaseModSq * pFunc) == \
...             ImplicitDiffusionTerm(diffusionCoeff) \
...             + sourceCoeff
```

If the example is run interactively, we create viewers for the phase and orientation variables.

```
>>> if __name__ == '__main__':
...     from fipy.viewers import make
...     phaseViewer = make(vars=phase,
...                         limits={'datamin': 0., 'datamax': 1.})
...     from fipy.tools.numerix import pi
...     thetaProductViewer = make(vars=theta,
...                               limits={'datamin': -pi,
...                                       'datamax': pi})
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

we iterate the solution in time, plotting as we go if running interactively,

```
>>> steps = 10
>>> for i in range(steps):
...     theta.updateOld()
...     phase.updateOld()
...     thetaEq.solve(theta, dt = timeStepDuration)
...     phaseEq.solve(phase, dt = timeStepDuration)
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

The solution is compared with test data. The test data was created with `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `test.gz` extracts the data and compares it with the `theta` variable.

```
>>> import os
>>> testFile = 'test.gz'
>>> import examples.phase.impingement.mesh40x1
```

```
>>> filepath = os.path.join(examples.phase.impingement.mesh40x1.__path__[0],
...                             testFile)
>>> import fipy.tools.dump as dump
>>> testData = dump.read(filepath)
>>> print theta.allclose(testData)
1
```

9.6   Module `examples.phase.impingement.mesh20x20.input`

In the following examples, we solve the same set of equations as in:

```
$ examples/phase/impingement/mesh40x1/input.py
```

with different initial conditions and a 2D mesh:

```
>>> from fipy.tools.parser import parse

>>> numberOfElements = parse('--numberOfElements', action = 'store',
...                             type = 'int', default = 400)
>>> numberOfSteps = parse('--numberOfSteps', action = 'store',
...                             type = 'int', default = 10)

>>> steps = numberOfSteps
>>> from fipy.tools.numerix import sqrt
>>> N = int(sqrt(numberOfElements))
>>> L = 2.5 * N / 100.
>>> dL = L / N
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx=dL, dy=dL, nx=N, ny=N)
```

The initial conditions are given by $\phi = 1$ and

$$
\theta = \begin{cases}
\frac{2\pi}{3} & \text{for } x^2 - y^2 < L/2, \\
\frac{-2\pi}{3} & \text{for } (x - L)^2 - y^2 < L/2, \\
\frac{-2\pi}{3} + 0.3 & \text{for } x^2 - (y - L)^2 < L/2, \\
\frac{2\pi}{3} & \text{for } (x - L)^2 - (y - L)^2 < L/2.
\end{cases}
$$

This defines four solid regions with different orientations. Solidification occurs and then boundary wetting occurs where the orientation varies.
The parameters for this example are

```
>>> timeStepDuration = 0.02
>>> phaseTransientCoeff = 0.1
>>> thetaSmallValue = 1e-6
>>> beta = 1e5
>>> mu = 1e3
>>> thetaTransientCoeff = 0.01
>>> gamma= 1e3
>>> epsilon = 0.008
>>> s = 0.01
>>> alpha = 0.015
```

The system is held isothermal at

```
>>> temperature = 10.
```

and is initialized to liquid everywhere

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phase = CellVariable(name='phase field', mesh=mesh)
```

The orientation is initialized to a uniform value to denote the randomly oriented liquid phase

```
>>> from fipy.variables.modularVariable import ModularVariable
>>> from fipy.tools.numerix import pi
>>> theta = ModularVariable(
...     name='theta',
...     mesh=mesh,
...     value=-pi + 0.0001,
...     hasOld=1
...     )
```

Four different solid circular domains are created at each corner of the domain with appropriate orientations

```
>>> x, y = mesh.getCellCenters()[...,0], mesh.getCellCenters()[...,1]
>>> for a, b, thetaValue in ((0., 0.,  2. * pi / 3.),
...                          (L, 0., -2. * pi / 3.),
...                          (0., L, -2. * pi / 3. + 0.3),
...                          (L, L,  2. * pi / 3.)):
...     segment = (x - a)**2 + (y - b)**2 < (L / 2.)**2
...     phase.setValue(1., where=segment)
...     theta.setValue(thetaValue, where=segment)
```

The `phase` equation is built in the following way. The source term is linearized in the manner demonstrated in `examples.phase.simple.input` (Kobayashi, semi-implicit). Here we use a function to build the equation, so that it can be reused later.

```
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.explicitDiffusionTerm import ExplicitDiffusionTerm
>>> from fipy.terms.implicitSourceTerm import ImplicitSourceTerm

>>> def buildPhaseEquation(phase, theta):
...
...     mPhiVar = phase - 0.5 + temperature * phase * (1 - phase)
...     thetaMag = theta.getOld().getGrad().getMag()
...     implicitSource = mPhiVar * (phase - (mPhiVar < 0))
...     implicitSource += (2 * s + epsilon**2 * thetaMag) * thetaMag
...
...     return TransientTerm(phaseTransientCoeff) == \
...             ExplicitDiffusionTerm(alpha**2) \
...             - ImplicitSourceTerm(implicitSource) \
...             + (mPhiVar > 0) * mPhiVar * phase

>>> phaseEq = buildPhaseEquation(phase, theta)
```

The `theta` equation is built in the following way. The details for this equation are fairly involved, see J.A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of `theta` on the circle. The source term requires the evaluation of the face gradient without the modular operators.

```
>>> from fipy.tools.numerix import exp
>>> def buildThetaEquation(phase, theta):
...
...         phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
...         phaseModSq = phaseMod * phaseMod
...         expo = epsilon * beta * theta.getGrad().getMag()
...         expo = (expo < 100.) * (expo - 100.) + 100.
...         pFunc = 1. + exp(-expo) * (mu / epsilon - 1.)
...
...         phaseFace = phase.getArithmeticFaceValue()
...         phaseSq = phaseFace * phaseFace
...         gradMag = theta.getFaceGrad().getMag()
...         eps = 1. / gamma / 10.
...         gradMag += (gradMag < eps) * eps
...         IGamma = (gradMag > 1. / gamma) * (1 / gradMag - gamma) + gamma
...         diffusionCoeff = phaseSq * (s * IGamma + epsilon**2)
...
...         thetaGradDiff = theta.getFaceGrad() - theta.getFaceGradNoMod()
...         sourceCoeff = (diffusionCoeff * thetaGradDiff).getDivergence()
...
...         from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
...         return TransientTerm(thetaTransientCoeff * phaseModSq * pFunc) == \
...                       ImplicitDiffusionTerm(diffusionCoeff) \
...                       + sourceCoeff

>>> thetaEq = buildThetaEquation(phase, theta)
```

If the example is run interactively, we create viewers for the phase and orientation variables. Rather than viewing the raw orientation, which is not meaningful in the liquid phase, we weight the orientation by the phase

```
>>> if __name__ == '__main__':
...         from fipy.viewers import make
...         phaseViewer = make(vars=phase,
...                            limits={'datamin': 0., 'datamax': 1.})
...         thetaProd = -pi + phase * (theta + pi)
...         thetaProductViewer = make(vars=thetaProd,
...                                   limits={'datamin': -pi,
...                                           'datamax': pi})
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

The solution will be tested against data that was created with `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `test.gz` extracts the data and compares it with the `theta` variable.

```
>>> import os
```

```
>>> testFile = 'test.gz'
>>> import examples.phase.impingement.mesh20x20
>>> import gzip
>>> filepath = os.path.join(examples.phase.impingement.mesh20x20.__path__[0],
...                         testFile)
>>> filestream = gzip.open(filepath,'r')
>>> import cPickle
>>> testData = cPickle.load(filestream)
>>> filestream.close()


>>> from fipy.tools.numerix import resize
>>> testData = resize(testData, (mesh.getNumberOfCells(),))
```

We step the solution in time, plotting as we go if running interactively,

```
>>> for i in range(steps):
...     theta.updateOld()
...     phase.updateOld()
...     thetaEq.solve(theta, dt=timeStepDuration)
...     phaseEq.solve(phase, dt=timeStepDuration)
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

The solution is compared against Ryo Kobayashi's test data

```
>>> print theta.allclose(testData, rtol=1e-7, atol=1e-7)
1
```

The following code shows how to restart a simulation from some saved data. First, reset the variables to their original values.

```
>>> phase.setValue(0)
>>> theta.setValue(-pi + 0.0001)
>>> x, y = mesh.getCellCenters()[...,0], mesh.getCellCenters()[...,1]
>>> for a, b, thetaValue in ((0., 0.,  2. * pi / 3.),
...                          (L, 0., -2. * pi / 3.),
...                          (0., L, -2. * pi / 3. + 0.3),
...                          (L, L,  2. * pi / 3.)):
...     segment = (x - a)**2 + (y - b)**2 < (L / 2.)**2
...     phase.setValue(1., where=segment)
...     theta.setValue(thetaValue, where=segment)
```

Step through half the time steps.

```
>>> for i in range(steps / 2):
...     theta.updateOld()
...     phase.updateOld()
...     thetaEq.solve(theta, dt=timeStepDuration)
...     phaseEq.solve(phase, dt=timeStepDuration)
```

We confirm that the solution has not yet converged to that given by Ryo Kobayashi's FORTRAN code:

```
>>> print theta.allclose(testData)
0
```

We save the variables to disk.

```
>>> from fipy.tools import dump
>>> (f, filename) = dump.write({'phase' : phase, 'theta' : theta}, extension = '.gz')
```

and then recall them to test the data pickling mechanism

```
>>> data = dump.read(filename, f)
>>> newPhase = data['phase']
>>> newTheta = data['theta']
>>> newThetaEq = buildThetaEquation(newPhase, newTheta)
>>> newPhaseEq = buildPhaseEquation(newPhase, newTheta)
```

and finish the iterations,

```
>>> for i in range(steps / 2):
...     newTheta.updateOld()
...     newPhase.updateOld()
...     newThetaEq.solve(newTheta, dt=timeStepDuration)
...     newPhaseEq.solve(newPhase, dt=timeStepDuration)
```

The solution is compared against Ryo Kobayashi's test data

```
>>> print newTheta.allclose(testData, rtol=1e-7)
1
```

Chapter 10

Level Set Examples

The Level Set Method (LSM) is a popular interface tracking method. Further details of the LSM and descriptions of the algorithms used in FiPy can be found in Sethian's Level Set book [32].

10.1 Module `examples.levelSet.distanceFunction.oneD.input`

Here we create a level set variable in one dimension. The level set variable calculates its value over the domain to be the distance from the zero level set. This can be represented succinctly in the following equation with a boundary condition at the zero level set such that,

$$\frac{\partial \phi}{\partial x} = 1$$

with the boundary condition, $\phi = 0$ at $x = L/2$. The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> dx = 0.5
>>> nx = 10
```

Construct the mesh.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx=dx, nx=nx)
```

Construct a `distanceVariable` object.

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable \
...     import DistanceVariable
>>> var = DistanceVariable(name='level set variable',
...                        mesh=mesh,
...                        value=-1,
...                        hasOld=1)
>>> x = mesh.getCellCenters()[...,0]
>>> var.setValue(1, where=x > dx * nx / 2)
```

Once the initial positive and negative regions have been initialized the `calcDistanceFunction()` method can be used to recalculate `var` as a distance function from the zero level set.

```
>>> var.calcDistanceFunction()
```

The problem can then be solved by executing the `solve()` method of the equation.

```
>>> if __name__ == '__main__':
...     from fipy.viewers import make
...     viewer = make(vars=var,
...                   limits={'datamin': -5., 'datamax': 5.})
...     viewer.plot()
```

The result can be tested with the following commands.

```
>>> from fipy.tools import numerix
>>> print numerix.allclose(var, x - dx * nx / 2)
1
```

10.2   Module `examples.levelSet.distanceFunction.circle.input`

Here we solve the level set equation in two dimensions for a circle. The 2D level set equation can be written,

$$|\nabla\phi| = 1$$

and the boundary condition for a circle is given by, $\phi = 0$ at $(x - L/2)^2 + (y - L/2)^2 = (L/4)^2$. The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = nx * dx
>>> Ly = ny * dy
```

Construct the mesh.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

Construct a `distanceVariable` object.

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable \
...     import DistanceVariable
>>> var = DistanceVariable(name='level set variable',
...                        mesh=mesh,
...                        value=-1,
...                        hasOld=1)

>>> x, y = mesh.getCellCenters()[...,0], mesh.getCellCenters()[...,1]
>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)

>>> var.calcDistanceFunction()
```

```
>>> if __name__ == '__main__':
...     from fipy import viewers
...     viewer = viewers.make(vars = var,
...                           limits = {'datamin': -5., 'datamax': 5.})
...     viewer.plot()
```

The result can be tested with the following commands.

```
>>> dY = dy / 2.
>>> dX = dx / 2.
>>> mm = min (dX, dY)
>>> from fipy.tools import numerix
>>> m1 = dY * dX / numerix.sqrt(dY**2 + dX**2)
>>> def evalCell(phix, phiy, dx, dy):
...     aa = dy**2 + dx**2
...     bb = -2 * ( phix * dy**2 + phiy * dx**2)
...     cc = dy**2 * phix**2 + dx**2 * phiy**2 - dx**2 * dy**2
...     sqr = numerix.sqrt(bb**2 - 4. * aa * cc)
...     return ((-bb - sqr) / 2. / aa,  (-bb + sqr) / 2. / aa)
>>> v1 = evalCell(-dY, -m1, dx, dy)[0]
>>> v2 = evalCell(-m1, -dX, dx, dy)[0]
>>> v3 = evalCell(m1,  m1,  dx, dy)[1]
>>> v4 = evalCell(v3, dY, dx, dy)[1]
>>> v5 = evalCell(dX, v3, dx, dy)[1]
>>> MASK = -1000
>>> trialValues = numerix.MA.masked_values((
...     MASK,  MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK,
...     MASK,  MASK, MASK, MASK,-3*dY,-3*dY,-3*dY, MASK, MASK, MASK, MASK,
...     MASK,  MASK, MASK,   v1,  -dY,  -dY,  -dY,   v1, MASK, MASK, MASK,
...     MASK,  MASK,   v2,  -m1,   m1,   dY,   m1,  -m1,   v2, MASK, MASK,
...     MASK, -dX*3,  -dX,   m1,   v3,   v4,   v3,   m1,  -dX,-dX*3, MASK,
...     MASK, -dX*3,  -dX,   dX,   v5, MASK,   v5,   dX,  -dX,-dX*3, MASK,
...     MASK, -dX*3,  -dX,   m1,   v3,   v4,   v3,   m1,  -dX,-dX*3, MASK,
...     MASK,  MASK,   v2,  -m1,   m1,   dY,   m1,  -m1,   v2, MASK, MASK,
...     MASK,  MASK, MASK,   v1,  -dY,  -dY,  -dY,   v1, MASK, MASK, MASK,
...     MASK,  MASK, MASK, MASK,-3*dY,-3*dY,-3*dY, MASK, MASK, MASK, MASK,
...     MASK,  MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK),
...     MASK)
>>> print var.allclose(trialValues)
1
```

10.3   Module `examples.levelSet.advection.mesh1D.input`

This example first solves the distance function equation in one dimension:

$$|\nabla\phi| = 1$$

with $\phi = 0$ at $x = L/5$. The variable is then advected with,

$$\frac{\partial\phi}{\partial t} + \vec{u} \cdot \nabla\phi = 0$$

The scheme used in the `AdvectionTerm` preserves the `var` as a distance function.
The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> velocity = 1.
>>> dx = 1.
>>> nx = 10
>>> timeStepDuration = 1.
>>> steps = 2
>>> L = nx * dx
>>> interfacePosition = L / 5.
```

Construct the mesh.

```
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx=dx, nx=nx)
```

Construct a `distanceVariable` object.

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable \
...     import DistanceVariable
>>> var = DistanceVariable(name='level set variable',
...                        mesh=mesh,
...                        value=-1,
...                        hasOld=1)
>>> var.setValue(1, where=mesh.getCellCenters()[...,0] > interfacePosition)
>>> var.calcDistanceFunction()
```

The `advectionEquation` is constructed.

```
>>> from fipy.models.levelSet.advection.advectionEquation import \
...     buildAdvectionEquation
>>> advEqn = buildAdvectionEquation(advectionCoeff=velocity)
```

The problem can then be solved by executing a serious of time steps.

```
>>> if __name__ == '__main__':
...     from fipy.viewers import make
...     viewer = make(vars=var,
...                   limits={'datamin': -10., 'datamax': 10.})
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
...         advEqn.solve(var, dt=timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following code:

```
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = mesh.getCellCenters()[:,0]
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = x - interfacePosition - timeStepDuration * steps * velocity
>>> from fipy.tools import numerix
```

```
>>> answer = numerix.where(x < distanceTravelled,
...                          x[0] - interfacePosition, answer)
>>> print var.allclose(answer)
1
```

10.4 Module `examples.levelSet.advection.circle.input`

This example first imposes a circular distance function:

$$\phi\left(x,y\right) = \left[\left(x - \frac{L}{2}\right)^2 + \left(y - \frac{L}{2}\right)^2\right]^{1/2} - \frac{L}{4}$$

The variable is advected with,

$$\frac{\partial \phi}{\partial t} + \vec{u} \cdot \nabla \phi = 0$$

The scheme used in the ˍAdvectionTerm preserves the `var` as a distance function. The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> L = 1.
>>> N = 25
>>> velocity = 1.
>>> cfl = 0.1
>>> velocity = 1.
>>> distanceToTravel = L / 10.
>>> radius = L / 4.
>>> dL = L / N
>>> timeStepDuration = cfl * dL / velocity
>>> steps = int(distanceToTravel / dL / cfl)
```

Construct the mesh.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx=dL, dy=dL, nx=N, ny=N)
```

Construct a `distanceVariable` object.

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable \
...      import DistanceVariable
>>> var = DistanceVariable(
...      name = 'level set variable',
...      mesh = mesh,
...      value = 1.,
...      hasOld = 1)
```

Initialise the `distanceVariable` to be a circular distance function.

```
>>> from fipy.tools import numerix
>>> initialArray = numerix.sqrt((mesh.getCellCenters()[:,0] - L / 2.)**2 +
...                              (mesh.getCellCenters()[:,1] - L / 2.)**2) - \
...                              radius
>>> var.setValue(initialArray)
```

The `advectionEquation` is constructed.

```
>>> from fipy.models.levelSet.advection.advectionEquation import \
...     buildAdvectionEquation
>>> advEqn = buildAdvectionEquation(
...     advectionCoeff=velocity)
```

The problem can then be solved by executing a serious of time steps.

```
>>> if __name__ == '__main__':
...     from fipy.viewers import make
...     viewer = make(vars=var,
...                   limits={'datamin': -radius, 'datamax': radius})
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
...         advEqn.solve(var, dt=timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following commands.

```
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = numerix.array(mesh.getCellCenters())
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = initialArray - distanceTravelled
>>> answer = numerix.where(answer < 0., -1001., answer)
>>> solution = numerix.where(answer < 0., -1001., numerix.array(var))
>>> numerix.allclose(answer, solution, atol=4.7e-3)
1
```

If the `AdvectionEquation` is built with the `_HigherOrderAdvectionTerm` the result is more accurate,

```
>>> var.setValue(initialArray)
>>> from fipy.models.levelSet.advection.higherOrderAdvectionEquation \
...     import buildHigherOrderAdvectionEquation
>>> advEqn = buildHigherOrderAdvectionEquation(
...     advectionCoeff = velocity)
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> solution = numerix.where(answer < 0., -1001., numerix.array(var))
>>> numerix.allclose(answer, solution, atol=1.02e-3)
1
```

Superconformal Electrodeposition Examples

The Damascene Process

State of the art manufacturing of semiconductor devices involves the electrodeposition of copper for on-chip wiring of integrated circuits. In the Damascene process interconnects are fabricated by first patterning trenches in a dielectric medium and then filling by metal electrodeposition over the entire wafer surface. This metalization process, pioneered by IBM, depends on the use of electrolyte additives that effect the local metal deposition rate.

Superfill

The additives in the electrolyte affect the local deposition rate in such a way that bottom-up filling occurs in trenches or vias. This process, known as superconformal electrodeposition or superfill, is demonstrated in the following figure. The figure shows sequential images of bottom-up superfilling of submicrometer trenches by copper deposition from an electrolyte containing PEG-SPS-Cl. Preferential metal deposition at the bottom of the trenches followed by bump formation above the filled trenches is evident.
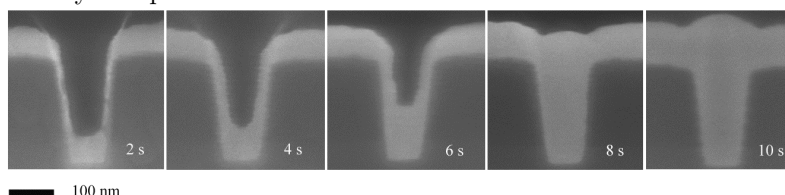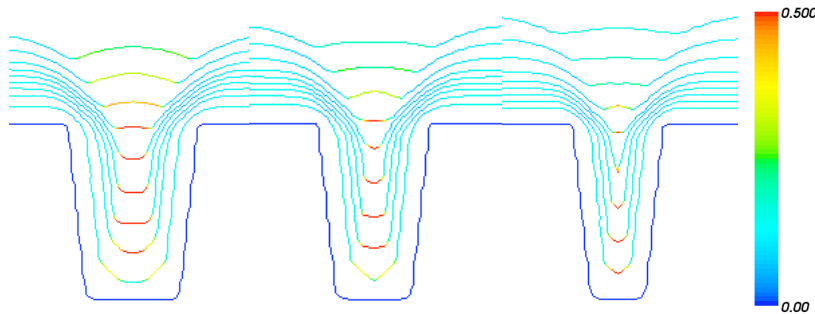

The CEAC Mechanism

This process has been demonstrated to depend critically on the inclusion of additives in the electrolyte. Recent publications propose Curvature Enhanced Accelerator Coverage (CEAC) as the mechanism behind the superfilling process [5]. In this mechanism, molecules that accelerate local metal deposition displace molecules that inhibit local metal deposition on the metal/electrolyte interface. For electrolytes that yield superconformal filling of fine features, this buildup happens relatively slowly because the concentration of accelerator species is much more dilute compared to the inhibitor species in the electrolyte. The mechanism that leads to the increased rate of metal deposition along the bottom of the filling trench is the concurrent local increase of the accelerator coverage due to decreasing local surface area, which scales with the local curvature (hence the name of the mechanism). A good overview of this mechanism can be found in [33].

Using FiPy to model Superfill

Example 10.5 provides a simple way to use FiPy to model the superfill process. The example includes a detailed description of the governing equations and feature geometry. It requires the user to import and execute a function at the python prompt. The model parameters can be passed as arguments to this function. In future all superfill examples will be provided with this type of interface. Example 10.8 has the same functionality as 10.5 but demonstrates how to write a new script in the case where the existing suite of scripts do not meet the required needs.

In general it is a good idea to obtain the Mayavi plotting package for which a specialized superfill viewer class has been created, see Chapter 2 "Installation and Usage". The other standard viewers mentioned in Chapter 2 "Installation and Usage" are still adequate although they do not give such clear images that are tailored for the superfill problem. The images below demonstrate the Mayavi viewing capability. Each contour represents sequential positions of the interface and the color represents the concentration of accelerator as a surfactant. The areas of high surfactant concentration have an increased deposition rate.

10.5   Module `examples.levelSet.electroChem.inputSimpleTrenchSystem`

This input file is a demonstration of the use of FiPy for modeling electrodeposition using the CEAC mechanism. The material properties and experimental parameters used are roughly those that have been previously published [34]. To run this example from the base fipy directory type:

```
$ examples/levelSet/electroChem/inputSimpleTrenchSystem.py
```

at the command line. The results of the simulation will be displayed and the word `finished` in the terminal at the end of the simulation. In order to alter the number of timesteps, the python function that encapsulates the system of equations must first be imported (at the python command line),

```
>>> from examples.levelSet.electroChem.inputSimpleTrenchSystem import runSimpleTrenchSystem
```

and then the function can be run with a different number of time steps with the `numberOfSteps` argument as follows,

```
>>> runSimpleTrenchSystem(numberOfSteps=5, displayViewers=False)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. Example 10.8 gives explanation for writing new scripts or modifying existing scripts that are encapsulated by functions. Any argument parameter can be changed. For example if the initial catalyst coverage is not 0, then it can be reset,

```
>>> runSimpleTrenchSystem(catalystCoverage=0.1, displayViewers=False)
0
```

The following image shows a schematic of a trench geometry along with the governing equations for modeling electrodeposition with the CEAC mechanism. All of the given equations are implemented in the `runSimpleTrenchSystem` function. As stated above, all the parameters in the equations can be changed with function arguments.
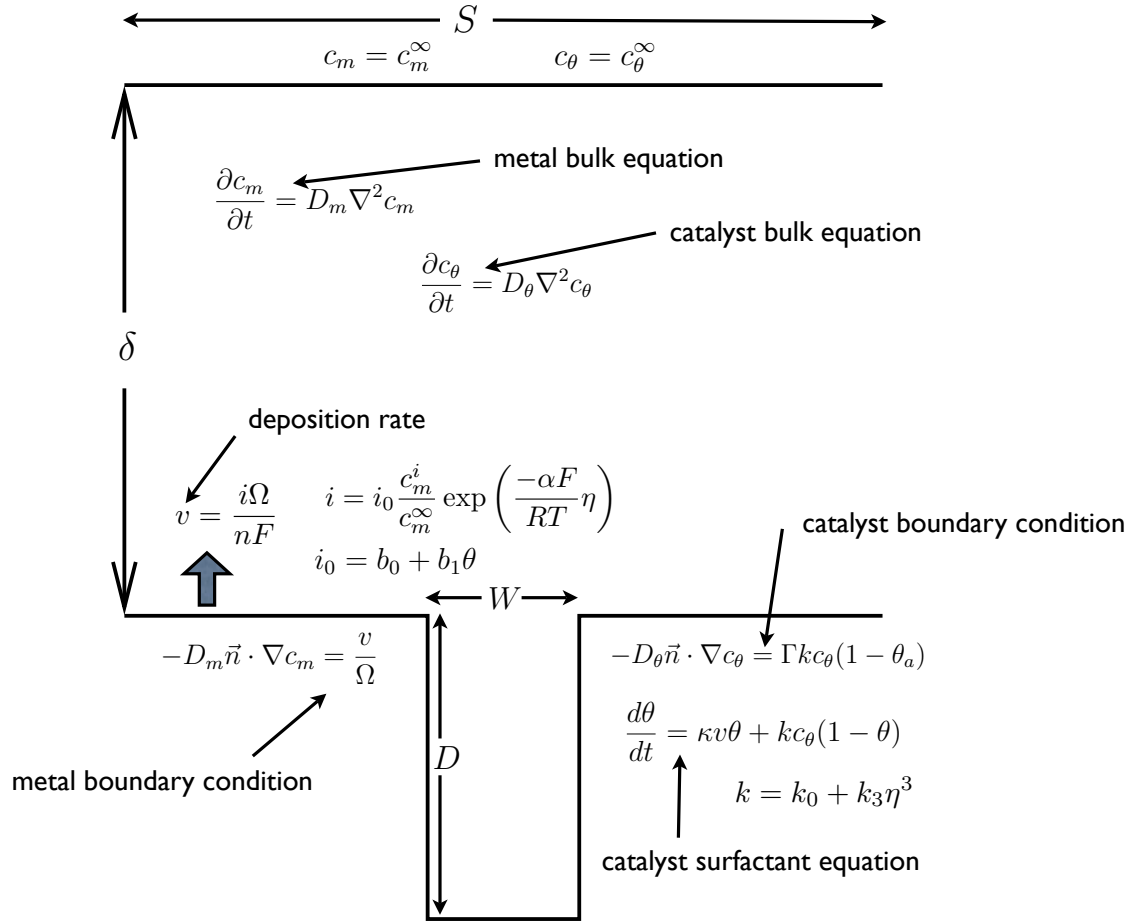
$$S$$

$$c_m = c_m^\infty \qquad c_\theta = c_\theta^\infty$$

metal bulk equation

$$\frac{\partial c_m}{\partial t} = D_m \nabla^2 c_m$$

catalyst bulk equation

$$\frac{\partial c_\theta}{\partial t} = D_\theta \nabla^2 c_\theta$$

$$\delta$$

deposition rate

$$v = \frac{i\Omega}{nF} \qquad i = i_0 \frac{c_m^i}{c_m^\infty} \exp\left(\frac{-\alpha F}{RT}\eta\right)$$

$$i_0 = b_0 + b_1\theta$$

catalyst boundary condition

$$W$$

$$-D_m \vec{n} \cdot \nabla c_m = \frac{v}{\Omega}$$

$$-D_\theta \vec{n} \cdot \nabla c_\theta = \Gamma k c_\theta (1 - \theta_a)$$

$$\frac{d\theta}{dt} = \kappa v \theta + k c_\theta (1 - \theta)$$

$$D$$

$$k = k_0 + k_3 \eta^3$$

metal boundary condition

catalyst surfactant equation

The following table shows the symbols used in the governing equations and their corresponding arguments to the `runSimpleTrenchSystem` function. The boundary layer depth is intentionally small in this example in order not to complicate the mesh. Further examples will simulate more realistic boundary layer depths but will also have more complex meshes requiring the `gmsh` software.

Symbol	Description	Keyword Argument	Value	Unit
		Deposition Rate Parameters		
v	deposition rate			m s^{-1}
i	current density			A m^{-2}
Ω	molar volume	`molarVolume`	7.1×10^{-6}	m^3 mol^{-1}
n	ion charge	`charge`	2	
F	Faraday's constant	`faradaysConstant`	9.6×10^{-4}	C mol^{-1}
i_0	exchange current density			A m^{-2}
α	transfer coefficient	`transferCoefficient`	0.5	
η	overpotential	`overpotential`	-0.3	V
R	gas constant	`gasConstant`	8.314	J K^{-1} mol^{-1}
T	temperature	`temperature`	298.0	K
b_0	current density for θ^0	`currentDensity0`	0.26	A m^{-2}
b_1	current density for θ	`currentDensity1`	45.0	A m^{-2}
		Metal Ion Parameters		
c_m	metal ion concentration	`metalConcentration`	250.0	mol m^{-3}
c_m^∞	far field metal ion concentration	`metalConcentration`	250.0	mol m^{-3}
D_m	metal ion diffusion coefficient	`metalDiffusion`	5.6×10^{-10}	m^2 s^{-1}
		Catalyst Parameters		
θ	catalyst surfactant concentration	`catalystCoverage`	0.0	
c_θ	bulk catalyst concentration	`catalystConcentration`	5.0×10^{-3}	mol m^{-3}
c_θ^∞	far field catalyst concentration	`catalystConcentration`	5.0×10^{-3}	mol m^{-3}
D_θ	catalyst diffusion coefficient	`catalystDiffusion`	1.0×10^{-9}	m^2 s^{-1}
Γ	catalyst site density	`siteDensity`	9.8×10^{-6}	mol m^{-2}
k	rate constant			m^3 mol^{-1} s^{-1}
k_0	rate constant for η^0	`rateConstant0`	1.76	m^3 mol^{-1} s^{-1}
k_3	rate constant for η^3	`rateConstant3`	-245.0×10^{-6}	m^3 mol^{-1} s^{-1} V^{-3}
		Geometry Parameters		
D	trench depth	`trenchDepth`	0.5×10^{-6}	m
D/W	trench aspect ratio	`aspectRatio`	2.0	
S	trench spacing	`trenchSpacing`	0.6×10^{-6}	m
δ	boundary layer depth	`boundaryLayerDepth`	0.3×10^{-6}	m
		Simulation Control Parameters		
	computational cell size	`cellSize`	0.1×10^{-7}	m
	number of time steps	`numberOfSteps`	5	
	whether to display the viewers	`displayViewers`	True	

If the MayaVi plotting software is installed (see Chapter 2) then a plot should appear that is updated every 20 time steps and will eventually resemble the image below.

catalyst_coverage

0.00   1.00
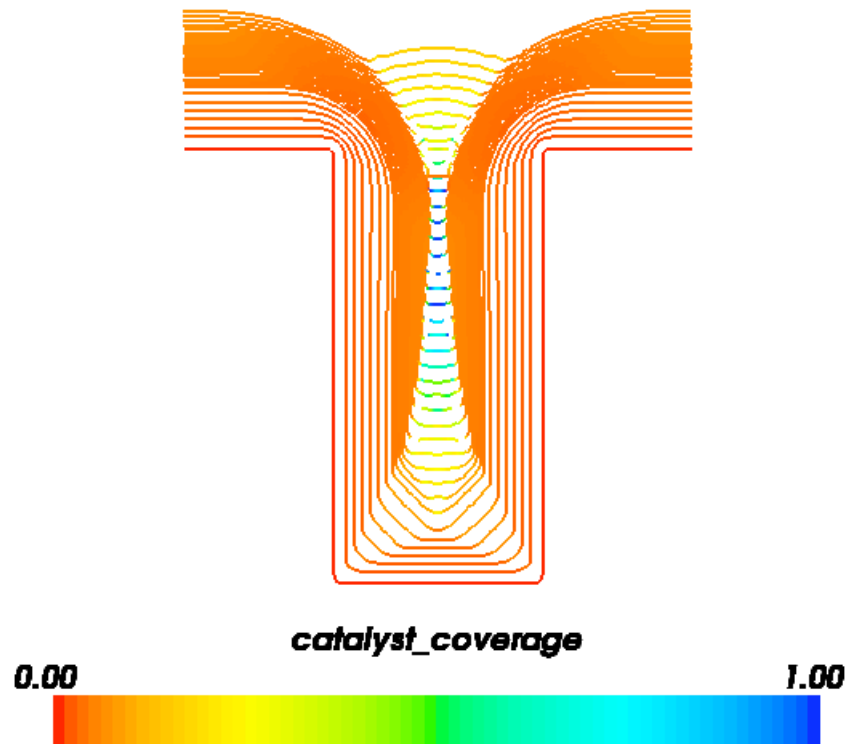
Functions

runSimpleTrenchSystem($faradaysConstant$=96000.0,
                       $gasConstant$=8.3140000000000001, $transferCoefficient$=0.5,
                       $rateConstant0$=1.76,
                       $rateConstant3$=-0.00024499999999999999,
                       $catalystDiffusion$=1.0000000000000001e-09,
                       $siteDensity$=9.7999999999999993e-06,
                       $molarVolume$=7.0999999999999998e-06, $charge$=2,
                       $metalDiffusion$=5.6000000000000003e-10, $temperature$=298.0,
                       $overpotential$=-0.29999999999999999,
                       $metalConcentration$=250.0,
                       $catalystConcentration$=0.0050000000000000001,
                       $catalystCoverage$=0.0, $currentDensity0$=0.26000000000000001,
                       $currentDensity1$=45.0, $cellSize$=1e-08,
                       $trenchDepth$=4.9999999999999998e-07, $aspectRatio$=2.0,
                       $trenchSpacing$=5.9999999999999997e-07,
                       $boundaryLayerDepth$=2.9999999999999999e-07,
                       $numberOfSteps$=5, $displayViewers$=True)

10.6   Module `examples.levelSet.electroChem.inputGold`

This input file is a demonstration of the use of FiPy for modeling gold superfill. The material properties and experimental parameters used are roughly those that have been previously published [35]. To run this example from the base fipy directory type:

```
$ examples/levelSet/electroChem/inputGold.py
```

at the command line. The results of the simulation will be displayed and the word `finished` in the terminal at the end of the simulation. The simulation will only run for 10 time steps. In order to alter the number of timesteps, the python function that encapsulates the system of equations must first be imported (at the python command line),

```
>>> from examples.levelSet.electroChem.inputGold import runGold
```

and then the function can be run with a different number of time steps with the `numberOfSteps` argument as follows,

```
>>> runGold(numberOfSteps=10, displayViewers=False)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. This example has a more realistic default boundary layer depth and thus requires `gmsh` to construct a more complex mesh.

There are a few differences between the gold superfill model presented in this example and Example 10.5. Most default values have changed to account for a different metal ion (gold) and catalyst (lead). In this system the catalyst is not present in the electrolyte but instead has a non-zero initial coverage. Thus quantities associated with bulk catalyst and catalyst accumulation are not defined. The current density is given by,

$$i = \frac{c_m}{c_m^\infty} \left(b_0 + b_1 \theta \right).$$

The more common representation of the current density includes an exponential part. Here it is buried in b_0 and b_1. The governing equation for catalyst evolution includes a term for catalyst consumption on the interface and is given by

$$\dot{\theta} = Jv\theta - k_c v\theta$$

where k_c is the consumption coefficient (`consumptionRateConstant`). The trench geometry is also given a slight taper, given by `taperAngle`.

If the MayaVi plotting software is installed (see Chapter 2) then a plot should appear that is updated every 10 time steps and will eventually resemble the image below.

catalyst_coverage

0.00   1.00
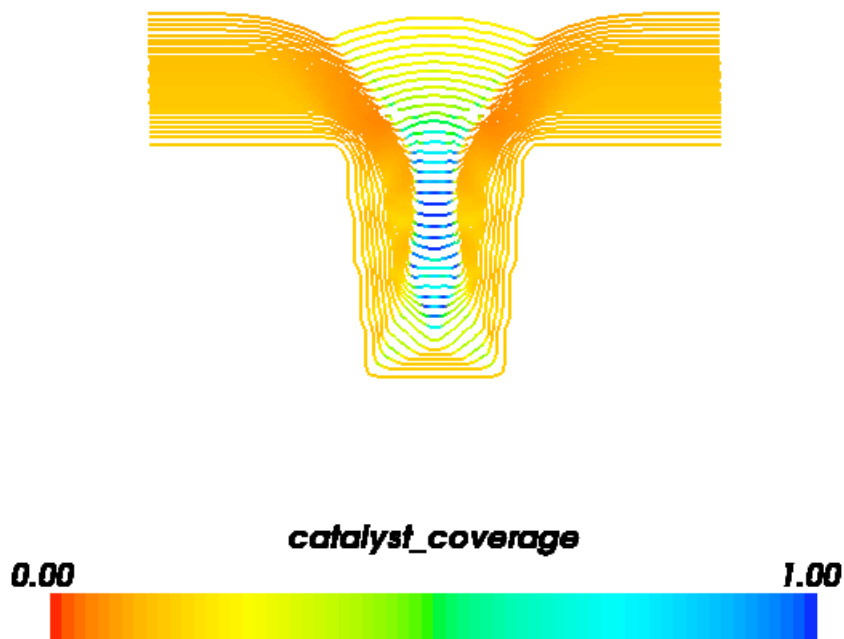
Functions

runGold(*faradaysConstant*=96000.0, *consumptionRateConstant*=2600000.0,
        molarVolume=1.0210000000000001e-05, *charge*=1.0,
        metalDiffusion=1.6999999999999999e-09, *metalConcentration*=20.0,
        catalystCoverage=0.14999999999999999,
        currentDensity0=0.47999999999999998, *currentDensity1*=10.4,
        cellSize=1e-08, *trenchDepth*=1.9999999999999999e-07, *aspectRatio*=1.47,
        trenchSpacing=4.9999999999999998e-07,
        boundaryLayerDepth=9.0000000000000006e-05, *numberOfSteps*=10,
        taperAngle=6.0, *displayViewers*=True)

10.7 Module examples.levelSet.electroChem.inputLeveler

This input file is a demonstration of the use of FiPy for modeling copper superfill with leveler and accelerator additives. The material properties and experimental parameters used are roughly those that have been previously published [36]. To run this example from the base fipy directory type:

    $ examples/levelSet/electroChem/inputLeveler.py

at the command line. The results of the simulation will be displayed and the word finished in the terminal at the end of the simulation. The simulation will only run for 200 time steps. In order to alter the number of timesteps, the python function that encapsulates the system of equations must first be imported (at the python command line),

```
>>> from examples.levelSet.electroChem.inputLeveler import runLeveler
```

and then the function can be executed with a different number of time steps by changing the `numberOfSteps` argument as follows,

```
>>> runLeveler(numberOfSteps=10, displayViewers=False, cellSize=0.25e-7)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. This example requires `gmsh` to construct the mesh.

This example models the case when suppressor, accelerator and leveler additives are present in the electrolyte. The suppressor is assumed to absorb quickly compared with the other additives. Any unoccupied surface sites are immediately covered with suppressor. The accelerator additive has more surface affinity than suppressor and is thus preferential adsorbed. The accelerator can also remove suppressor when the surface reaches full coverage. Similarly, the leveler additive has more surface affinity than both the suppressor and accelerator. This forms a simple set of assumptions for understanding the behavior of these additives.

The following is a complete description of the equations for the model described here. Any equations that have been omitted are the same as those given in Example 10.5. The current density is governed by

$$i = \frac{c_m}{c_m^\infty} \sum_j \left[i_j \theta_j \left(\exp \frac{-\alpha_j F \eta}{RT} - \exp \frac{(1 - \alpha_j) F \eta}{RT} \right) \right]$$

where j represents S for suppressor, A for accelerator, L for leveler and V for vacant. This model assumes a linear interpolation between the three cases of complete coverage for each additive or vacant substrate. The governing equations for the surfactants are given by,

$$\dot{\theta_L} = \kappa v \theta_L + k_l^+ c_L \left(1 - \theta_L \right) - k_L^- v \theta_L,$$

$$\dot{\theta_a} = \kappa v \theta_A + k_A^+ c_A \left(1 - \theta_A - \theta_L \right) - k_L c_L \theta_A - k_A^- \theta_A^{q-1},$$

$$\theta_S = 1 - \theta_A - \theta_L$$

and

$$\theta_V = 0.$$

It has been found experimentally that $i_L = i_S$.

If the surface reaches full coverage, the equations do not naturally prevent the coverage rising above full coverage due to the curvature terms. Thus, when $\theta_L + \theta_A = 1$ then the equation for accelerator becomes $\dot{\theta_A} = -\dot{\theta_L}$ and when $\theta_L = 1$, the equation for leveler becomes $\dot{\theta_L} = -k_L^- v \theta_L$.

The parameters k_A^+, k_A^- and q are both functions of η given by,

$$k_A^+ = k_{A0}^+ \exp \frac{-\alpha_k F \eta}{RT},$$

and

$$k_A^- = B_d + \frac{A}{\exp \left(B_a \left(\eta + V_d \right) \right)} + \exp \left(B_b \left(\eta + V_d \right) \right)$$

and

$$q = m * \eta + b.$$

The following table shows the symbols used in the governing equations and their corresponding arguments for the **runLeveler** function.

Symbol	Description	Keyword Argument	Value	Unit
		Deposition Rate Parameters		
v	deposition rate			m s^{-1}
i_A	accelerator current density	`iOAccelerator`		A m^{-2}
i_L	leveler current density	`iOLeveler`		A m^{-2}
Ω	molar volume	`molarVolume`	7.1×10^{-6}	m^3 mol^{-1}
n	ion charge	`charge`	2	
F	Faraday's constant	`faradaysConstant`	9.6×10^{-4}	C mol^{-1}
i_0	exchange current density			A m^{-2}
α_A	accelerator transfer coefficient	`alphaAccelerator`	0.4	
α_S	leveler transfer coefficient	`alphaLeveler`	0.5	
η	overpotential	`overpotential`	-0.3	V
R	gas constant	`gasConstant`	8.314	J K mol^{-1}
T	temperature	`temperature`	298.0	K
		Ion Parameters		
c_I	ion concentration	`ionConcentration`	250.0	mol m^{-3}
c_I^∞	far field ion concentration	`ionConcentration`	250.0	mol m^{-3}
D_I	ion diffusion coefficient	`ionDiffusion`	5.6×10^{-10}	m^2 s^{-1}
		Accelerator Parameters		
θ_A	accelerator coverage	`acceleratorCoverage`	0.0	
c_A	accelerator concentartion	`acceleratorConcentration`	5.0×10^{-3}	mol m^{-3}
c_A^∞	far field accelerator concentration	`acceleratorConcentration`	5.0×10^{-3}	mol m^{-3}
D_A	catalyst diffusion coefficient	`catalystDiffusion`	1.0×10^{-9}	m^2 s^{-1}
Γ_A	accelerator site density	`siteDensity`	9.8×10^{-6}	mol m^{-2}
k_A^+	accelerator adsorption			m^3 mol^{-1} s^{-1}
k_{A0}^+	accelerator adsorption coeff	`kAccelerator0`	$2.6times10^{-4}$	m^3 mol^{-1} s^{-1}
α_k	accelerator adsorption coeff	`alphaAdsorption`	0.62	
k_A^-	accelerator consumption coeff			
B_a	experimental parameter	`Bd`	-40.0	
B_b	experimental parameter	`Bd`	60.0	
V_d	experimental parameter	`Bd`	9.8×10^{-2}	
B_d	experimental parameter	`Bd`	8.0×10^{-4}	
		Geometry Parameters		
D	trench depth	`trenchDepth`	0.5×10^{-6}	m
D/W	trench aspect ratio	`aspectRatio`	2.0	
S	trench spacing	`trenchSpacing`	0.6×10^{-6}	m
δ	boundary layer depth	`boundaryLayerDepth`	0.3×10^{-6}	m
		Simulation Control Parameters		
	computational cell size	`cellSize`	0.1×10^{-7}	m
	number of time steps	`numberOfSteps`	5	
	whether to display the viewers	`displayViewers`	`True`	

The following images show accelerator and leveler contour plots that can be obtained by running this

example.

leveler_coverage

| 0.00 | 0.0714 | 0.143 | 0.214 | 0.286 | 0.357 | 0.429 | 0.600 |
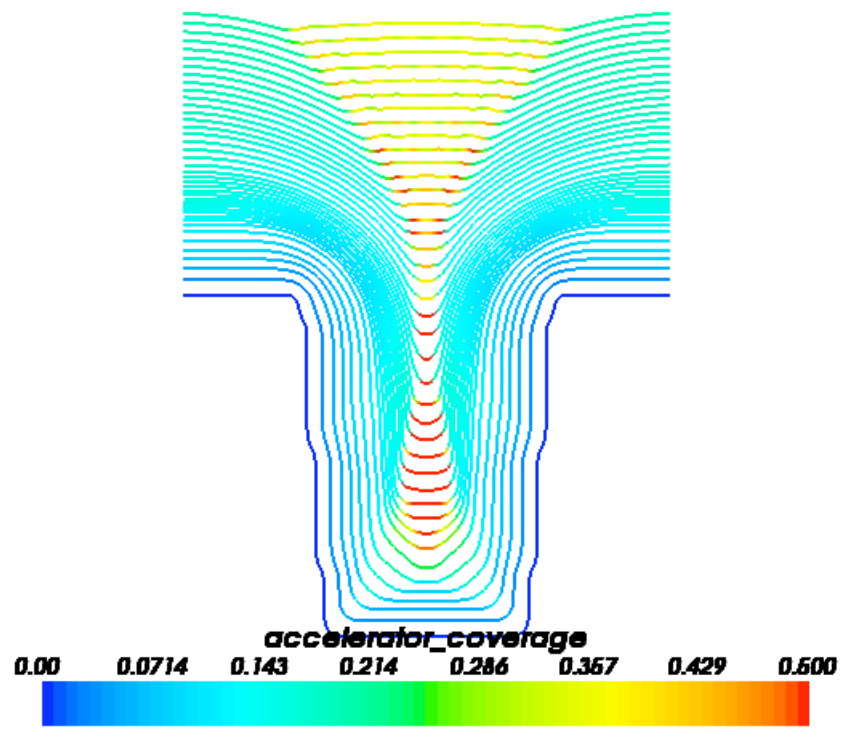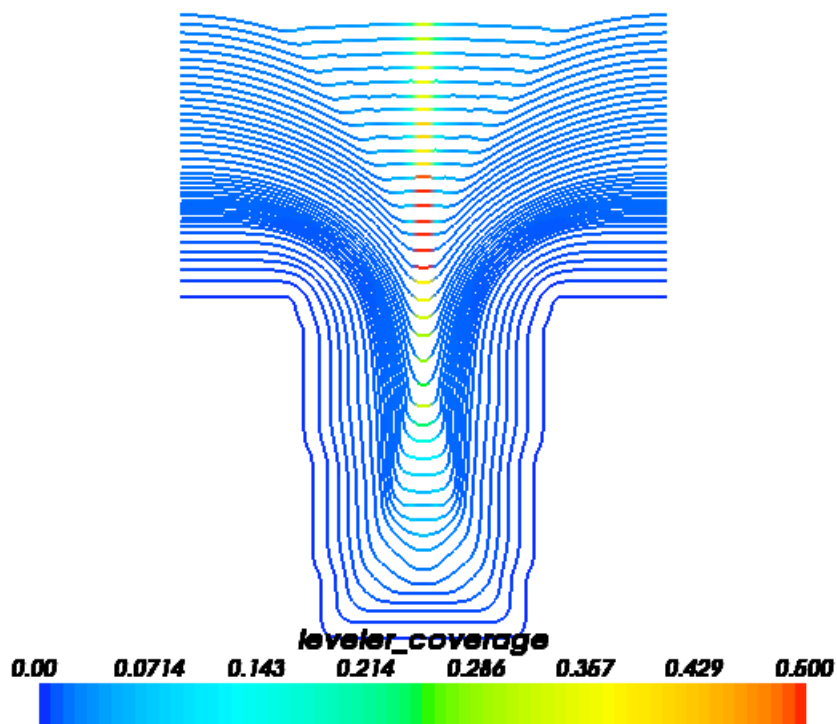
Functions

runLeveler(*kLeveler*=0.017999999999999999, *bulkLevelerConcentration*=0.02,
        cellSize=1e-08, *rateConstant*=0.00025999999999999998,
        initialAcceleratorCoverage=0.0,
        levelerDiffusionCoefficient=5.0000000000000003e-10, *numberOfSteps*=400,
        displayRate=10, *displayViewers*=True)

10.8   Module examples.levelSet.electroChem.inputWriteScriptHowTo

This input file demonstrates how to create a new superfill script if the existing suite of scripts do not meet the required needs. It provides the functionality of Example 10.5. To run this example from the base fipy directory type:

    $ examples/levelSet/electroChem/input.py --numberOfElements=10000 --numberOfSteps=800

at the command line. The results of the simulation will be displayed and the word finished in the terminal at the end of the simulation. To obtain this example in a plain script file in order to edit and run type:

    $ python setup.py copy_script --From examples/levelSet/electroChem/input.py --To myScript.py

in the base FiPy directory. The file `myScript.py` will contain the script.

The following is an explicit explanation of the input commands required to set up and run the problem. At the top of the file all the parameter values are set. Their use will be explained during the instantiation of various objects and are the same as those explained in Example 10.5. The following parameters (all in S.I. units) represent,

physical constants,

```
>>> faradaysConstant = 9.6e4
>>> gasConstant = 8.314
>>> transferCoefficient = 0.5
```

properties associated with the catalyst species,

```
>>> rateConstant0 = 1.76
>>> rateConstant3 = -245e-6
>>> catalystDiffusion = 1e-9
>>> siteDensity = 9.8e-6
```

properties of the cupric ions,

```
>>> molarVolume = 7.1e-6,
>>> charge = 2
>>> metalDiffusionCoefficient = 5.6e-10
```

parameters dependent on experimental constraints,

```
>>> temperature = 298.
>>> overpotential = -0.3
>>> bulkMetalConcentration = 250.
>>> catalystConcentration = 5e-3
>>> catalystCoverage = 0.
```

parameters obtained from experiments on flat copper electrodes,

```
>>> currentDensity0 = 0.26
>>> currentDensity1 = 45.
```

general simulation control parameters,

```
>>> cflNumber = 0.2
>>> numberOfCellsInNarrowBand = 10
>>> cellsBelowTrench = 10
>>> cellSize = 0.1e-7
```

parameters required for a trench geometry,

```
>>> trenchDepth = 0.5e-6
>>> aspectRatio = 2.
>>> trenchSpacing = 0.6e-6
>>> boundaryLayerDepth = 0.3e-6
```

The hydrodynamic boundary layer depth (`boundaryLayerDepth`) is intentionally small in this example to keep the mesh at a reasonable size.

Build the mesh:

```
>>> from fipy.tools.parser import parse
>>> numberOfElements = parse('--numberOfElements', action='store',
...     type='int', default=-1)
>>> numberOfSteps = parse('--numberOfSteps', action='store',
...     type='int', default=5)

>>> from fipy.tools.numerix import sqrt, exp, max
>>> if numberOfElements != -1:
...     pos = trenchSpacing * cellsBelowTrench / 4 / numberOfElements
...     sqr = trenchSpacing * (trenchDepth + boundaryLayerDepth) \
...             / (2 * numberOfElements)
...     cellSize = pos + sqrt(pos**2 + sqr)
... else:
...     cellSize = 0.1e-7

>>> yCells = cellsBelowTrench \
...           + int((trenchDepth + boundaryLayerDepth) / cellSize)
>>> xCells = int(trenchSpacing / 2 / cellSize)


>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx=cellSize,
...               dy=cellSize,
...               nx=xCells,
...               ny=yCells)
```

A `distanceVariable` object, ϕ, is required to store the position of the interface . The `distanceVariable` calculates its value so that it is a distance function (*i.e.* holds the distance at any point in the mesh from the electrolyte/metal interface at $\phi = 0$) and $|\nabla\phi| = 1$.
First, create the ϕ variable, which is initially set to -1 everywhere. Create an initial variable,

```
>>> narrowBandWidth = numberOfCellsInNarrowBand * cellSize
>>> from fipy.models.levelSet.distanceFunction.distanceVariable import \
...     DistanceVariable
>>> distanceVar = DistanceVariable(
...     name='distance variable',
...     mesh= mesh,
...     value=-1,
...     narrowBandWidth=narrowBandWidth,
...     hasOld=1)
```

The electrolyte region will be the positive region of the domain while the metal region will be negative.

```
>>> bottomHeight = cellsBelowTrench * cellSize
>>> trenchHeight = bottomHeight + trenchDepth
>>> trenchWidth = trenchDepth / aspectRatio
>>> sideWidth = (trenchSpacing - trenchWidth) / 2

>>> x, y = mesh.getCellCenters()[...,0], mesh.getCellCenters()[...,1]
>>> distanceVar.setValue(1, where=(y > trenchHeight)
...                                 | ((y > bottomHeight)
...                                     & (x < xCells * cellSize - sideWidth)))
```

```
>>> distanceVar.calcDistanceFunction(narrowBandWidth=1e10)
```

The `distanceVariable` has now been created to mark the interface. Some other variables need to be created that govern the concentrations of various species. Create the catalyst surfactant coverage, θ, variable. This variable influences the deposition rate.

..
¿¿¿ from fipy.models.levelSet.surfactant.surfactantVariable import ... SurfactantVariable ¿¿¿ catalystVar = SurfactantVariable(... name="catalyst variable", ... value=catalystCoverage, ... distanceVar=distanceVar)Create the bulk catalyst concentration, c_θ, in the electrolyte,

```
>>> from fipy.variables.cellVariable import CellVariable
>>> bulkCatalystVar = CellVariable(
...      name='bulk catalyst variable',
...      mesh=mesh,
...      value=catalystConcentration)
```

Create the bulk metal ion concentration, c_m, in the electrolyte.

```
>>> metalVar = CellVariable(
...      name='metal variable',
...      mesh=mesh,
...      value=bulkMetalConcentration)
```

The following commands build the `depositionRateVariable`, v. The `depositionRateVariable` is given by the following equation.

$$v = \frac{i\Omega}{nF}$$

where Ω is the metal molar volume, n is the metal ion charge and F is Faraday's constant. The current density is given by

$$i = i_0 \frac{c_m^i}{c_m^\infty} \exp\left(\frac{-\alpha F}{RT}\eta\right)$$

where c_m^i is the metal ion concentration in the bulk at the interface, c_m^∞ is the far-field bulk concentration of metal ions, α is the transfer coefficient, R is the gas constant, T is the temperature and η is the overpotential. The exchange current density is an empirical function of catalyst coverage,

$$i_0(\theta) = b_0 + b_1\theta$$

The commands needed to build this equation are,

```
>>> expoConstant = -transferCoefficient * faradaysConstant \
...                 / (gasConstant * temperature)
>>> tmp = currentDensity1 \
...         * catalystVar.getInterfaceVar()
>>> exchangeCurrentDensity = currentDensity0 + tmp
>>> expo = exp(expoConstant * overpotential)
>>> currentDensity = expo * exchangeCurrentDensity * metalVar \
...                   / bulkMetalConcentration
>>> depositionRateVariable = currentDensity * molarVolume \
...                           / (charge * faradaysConstant)
```

Build the extension velocity variable v_{ext}. The extension velocity uses the `extensionEquation` to spread the velocity at the interface to the rest of the domain.

```
>>> extensionVelocityVariable = CellVariable(
...       name='extension velocity',
...       mesh=mesh,
...       value=depositionRateVariable)
```

Using the variables created above the governing equations will be built. The governing equation for surfactant conservation is given by,

$$\dot{\theta} = Jv\theta + kc_\theta^i(1 - \theta)$$

where θ is the coverage of catalyst at the interface, J is the curvature of the interface, v is the normal velocity of the interface, c_θ^i is the concentration of catalyst in the bulk at the interface. The value k is given by an empirical function of overpotential,

$$k = k_0 + k_3\eta^3$$

The above equation is represented by the `AdsorbingSurfactantEquation` in FiPy:

```
>>> from fipy.models.levelSet.surfactant.adsorbingSurfactantEquation \
...             import AdsorbingSurfactantEquation
>>> surfactantEquation = AdsorbingSurfactantEquation(
...       surfactantVar=catalystVar,
...       distanceVar=distanceVar,
...       bulkVar=bulkCatalystVar,
...       rateConstant=rateConstant0 \
...                   + rateConstant3 * overpotential**3)
```

The variable ϕ is advected by the `advectionEquation` given by,

$$\frac{\partial \phi}{\partial t} + v_{\text{ext}}|\nabla\phi| = 0$$

and is set up with the following commands:

```
>>> from fipy.models.levelSet.advection.higherOrderAdvectionEquation \
...             import buildHigherOrderAdvectionEquation
>>> advectionEquation = buildHigherOrderAdvectionEquation(
...       advectionCoeff=extensionVelocityVariable)
```

The diffusion of metal ions from the far field to the interface is governed by,

$$\frac{\partial c_m}{\partial t} = \nabla \cdot D\nabla c_m$$

where,

$$D = \begin{cases} D_m & \text{when } \phi > 0, \\ 0 & \text{when } \phi \leq 0 \end{cases}$$

The following boundary condition applies at $\phi = 0$,

$$D\hat{n} \cdot \nabla c = \frac{v}{\Omega}.$$

The `MetalIonDiffusionEquation` is set up with the following commands.

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.models.levelSet.electroChem.metalIonDiffusionEquation \
...                      import buildMetalIonDiffusionEquation
>>> metalEquation = buildMetalIonDiffusionEquation(
...     ionVar=metalVar,
...     distanceVar=distanceVar,
...     depositionRate=depositionRateVariable,
...     diffusionCoeff=metalDiffusionCoefficient,
...     metalIonMolarVolume=molarVolume,
... )

>>> metalEquationBCs = FixedValue(faces=mesh.getFacesTop(), value=bulkMetalConcentration)
```

The `SurfactantBulkDiffusionEquation` solves the bulk diffusion of a species with a source term for the jump from the bulk to an interface. The governing equation is given by,

$$\frac{\partial c}{\partial t} = \nabla \cdot D \nabla c$$

where,

$$D = \begin{cases} D_\theta & \text{when } \phi > 0 \\ 0 & \text{when } \phi \le 0 \end{cases}$$

The jump condition at the interface is defined by Langmuir adsorption. Langmuir adsorption essentially states that the ability for a species to jump from an electrolyte to an interface is proportional to the concentration in the electrolyte, available site density and a jump coefficient. The boundary condition at $\phi = 0$ is given by,

$$D\hat{n} \cdot \nabla c = -kc(1 - \theta).$$

The `SurfactantBulkDiffusionEquation` is set up with the following commands.

```
>>> from fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation \
...                 import buildSurfactantBulkDiffusionEquation
>>> bulkCatalystEquation = buildSurfactantBulkDiffusionEquation(
...     bulkVar=bulkCatalystVar,
...     distanceVar=distanceVar,
...     surfactantVar=catalystVar,
...     diffusionCoeff=catalystDiffusion,
...     rateConstant=rateConstant0 * siteDensity
... )

>>> catalystBCs = FixedValue(faces=mesh.getFacesTop(), value=catalystConcentration)
```

If running interactively, create viewers.

```
>>> if __name__ == '__main__':
...     try:
...         from fipy.viewers.mayaviViewer.mayaviSurfactantViewer import MayaviSurfactantViewer
...         viewers = (
...             MayaviSurfactantViewer(distanceVar,
...                                    catalystVar.getInterfaceVar(),
```

```
...                                               zoomFactor=1e6,
...                                               limits={ 'datamax' : 1.0, 'datamin' : 0.0 },
...                                               smooth=1),)
...        except:
...            from fipy.viewers import make
...            viewers = (
...                make(distanceVar, limits={ 'datamin' :-1e-9 , 'datamax' : 1e-9 }),
...                make(catalystVar.getInterfaceVar()))
... else:
...        viewers = ()
```

The `levelSetUpdateFrequency` defines how often to call the `distanceEquation` to reinitialize the `distanceVariable` to a distance function.

```
>>> levelSetUpdateFrequency = int(0.8 * narrowBandWidth \
...                               / (cellSize * cflNumber * 2))
```

The following loop runs for `numberOfSteps` time steps. The time step is calculated with the CFL number and the maximum extension velocity. v to v_{ext} throughout the whole domain using $\nabla\phi \cdot \nabla v_{\mathrm{ext}} = 0$.

```
>>> for step in range(numberOfSteps):
...
...        for viewer in viewers:
...            viewer.plot()
...
...        if step % levelSetUpdateFrequency == 0:
...            distanceVar.calcDistanceFunction()
...
...        extensionVelocityVariable.setValue(depositionRateVariable())
...
...        distanceVar.updateOld()
...        catalystVar.updateOld()
...        metalVar.updateOld()
...        bulkCatalystVar.updateOld()
...        distanceVar.extendVariable(extensionVelocityVariable)
...        dt = cflNumber * cellSize / max(extensionVelocityVariable)
...        advectionEquation.solve(distanceVar, dt=dt)
...        surfactantEquation.solve(catalystVar, dt=dt)
...        metalEquation.solve(var=metalVar, dt=dt,
...                            boundaryConditions=metalEquationBCs)
...        bulkCatalystEquation.solve(var=bulkCatalystVar, dt=dt,
...                                   boundaryConditions=catalystBCs)

>>> if __name__ == '__main__':
...        raw_input('finished')
```

The following is a short test case. It uses saved data from a simulation with 5 time steps. It is not a test for accuracy but a way to tell if something has changed or been broken.

```
>>> import os
>>> import examples.levelSet.electroChem
>>> filepath = os.path.join(examples.levelSet.electroChem.__path__[0],
...                         'test.gz')
```

```
>>> from fipy.tools import dump
>>> print catalystVar.allclose(dump.read(filepath), rtol=1e-4)
1
```

Chapter 11

Cahn-Hilliard Examples

11.1 Module `examples.cahnHilliard.inputTanh1D`

This example solves the Cahn-Hilliard equation given by,

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right)$$

where the free energy functional is given by,

$$f = \frac{a^2}{2} \phi^2 (1 - \phi)^2$$

The Cahn-Hilliard equation can be rewritten in the following form,

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \left(\frac{\partial^2 f}{\partial \phi^2} \nabla \phi - \epsilon^2 \nabla^3 \phi \right)$$

The above form of the equation makes the non-linearity part of the diffusion coefficient for the first term on the RHS. This is the correct way to express the equation to FiPy. We solve the problem on a 1D mesh

```
>>> L = 40.
>>> nx = 1000
>>> dx = L / nx
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx=dx, nx=nx)
```

and create the solution variable

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(
...     name="phase field",
...     mesh=mesh,
...     value=1)
```

The boundary conditions for this problem are

$$\left. \begin{array}{r} \phi = \dfrac{1}{2} \\[2ex] \dfrac{\partial^2 \phi}{\partial x^2} = 0 \end{array} \right\} \quad \text{on } x = 0$$

and

$$\left.\begin{array}{r} \phi = 1 \\[1mm] \dfrac{\partial^2 \phi}{\partial x^2} = 0 \end{array}\right\} \qquad \text{on } x = L$$

or

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.nthOrderBoundaryCondition \
...     import NthOrderBoundaryCondition
>>> BCs = (
...     FixedValue(faces=mesh.getFacesRight(), value=1),
...     FixedValue(faces=mesh.getFacesLeft(), value=.5),
...     NthOrderBoundaryCondition(faces=mesh.getFacesLeft(), value=0, order=2),
...     NthOrderBoundaryCondition(faces=mesh.getFacesRight(), value=0, order=2))
```

Using

```
>>> asq = 1.0
>>> epsilon = 1
>>> diffusionCoeff = 1
```

we create the Cahn-Hilliard equation:

```
>>> faceVar = var.getArithmeticFaceValue()
>>> freeEnergyDoubleDerivative = asq * ( 1 - 6 * faceVar * (1 - faceVar))


>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> from fipy.terms.transientTerm import TransientTerm
>>> diffTerm2 = ImplicitDiffusionTerm(
...     coeff=diffusionCoeff * freeEnergyDoubleDerivative)
>>> diffTerm4 = ImplicitDiffusionTerm(coeff=(diffusionCoeff, epsilon**2))
>>> eqch = TransientTerm() == diffTerm2 - diffTerm4


>>> from fipy.solvers.linearLUSolver import LinearLUSolver
>>> solver = LinearLUSolver(tolerance=1e-15, iterations=100)
```

The solution to this 1D problem over an infinite domain is given by,

$$\phi(x) = \frac{1}{1 + \exp\left(-\frac{a}{\epsilon}x\right)}$$

or

```
>>> from fipy.tools import numerix
>>> a = numerix.sqrt(asq)
>>> answer = 1 / (1 +
...     numerix.exp(-a * (mesh.getCellCenters()[:,0]) / epsilon))
```

If we are running interactively, we create a viewer to see the results

```
>>> if __name__ == '__main__':
...     from fipy.viewers import make
...     viewer = make(vars=var,
...                   limits={'datamin': 0., 'datamax': 1.0})
...     viewer.plot()
```

We iterate the solution to equilibrium and, if we are running interactively, we update the display and output data about the progression of the solution

```
>>> dexp=-5
>>> for step in range(100):
...     dt = numerix.exp(dexp)
...     dt = min(10, dt)
...     dexp += 0.5
...     eqch.solve(var=var, boundaryConditions=BCs, solver=solver, dt=dt)
...     if __name__ == '__main__':
...         diff = abs(answer - numerix.array(var))
...         maxarg = numerix.argmax(diff)
...         print 'maximum error:',diff[maxarg]
...         print 'element id:',maxarg
...         print 'value at element ',maxarg,' is ',var[maxarg]
...         print 'solution value',answer[maxarg]
...
...         viewer.plot()
```

We compare the analytical solution with the numerical result,

```
>>> print var.allclose(answer, atol=1e-4)
1
```

Chapter 12

Fluid Flow Examples

12.1 Module `examples.flow.stokesCavity`

This example is an implementation of a rudimentary Stokes solver. It solves the Navier-Stokes equation in the viscous limit,

$$\nabla\mu \cdot \nabla\vec{u} = \nabla p$$

and the continuity equation,

$$\nabla \cdot \vec{u} = 0$$

where $\vec{u}$ is the fluid velocity, p is the pressure and μ is the viscosity. The domain in this example is a square cavity of unit dimensions with a moving lid of unit speed. This example uses the SIMPLE algorithm with Rhie-Chow interpolation to solve the pressure-momentum coupling. Some of the details of the algorithm will be highlighted below but a good reference for this material is Ferziger and Perić [37]. The solution has a high degree of error close to the corners of the domain for the pressure but does a reasonable job of predicting the velocities away from the boundaries. A number of aspects of FiPy need to be improved to have a first class flow solver. These include, higher order spatial diffusion terms, proper wall boundary conditions, improved mass flux evaluation and extrapolation of cell values to the boundaries using gradients. In the table below a comparison is made with the Dolfyn open source code on a 100 by 100 grid. The table shows the frequency of values that fall within the given error confidence bands. Dolfyn has the added features described above. When these features are switched off the results of Dolfyn and FiPy are identical.

% frequency of cells	x-velocity error (%)	y-velocity error (%)	pressure error (%)
90	< 0.1	< 0.1	< 5
5	0.1 to 0.6	0.1 to 0.3	5 to 11
4	0.6 to 7	0.3 to 4	11 to 35
1	7 to 96	4 to 80	35 to 179
0	> 96	> 80	> 179

To start, some parameters are declared.

```
>>> L = 1.0
>>> N = 50
>>> dL = L / N
>>> viscosity = 1.
>>> pressureRelaxation = 0.2
>>> velocityRelaxation = 0.5
>>> if __name__ == '__main__':
...     sweeps = 300
```

```
... else:
...       sweeps = 5
```

Build the mesh.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(nx=N, ny=N, dx=dL, dy=dL)
```

Declare the variables.

```
>>> from fipy.variables.cellVariable import CellVariable
>>> pressure = CellVariable(mesh=mesh, name='pressure')
>>> pressureCorrection = CellVariable(mesh=mesh)
>>> xVelocity = CellVariable(mesh=mesh, name='X velocity')
>>> yVelocity = CellVariable(mesh=mesh, name='Y velocity')
```

The velocity is required as a `VectorFaceVariable` for calculating the mass flux. This is a somewhat clumsy aspect of the FiPy interface that needs improvement.

```
>>> from fipy.variables.vectorFaceVariable import VectorFaceVariable
>>> velocity = VectorFaceVariable(mesh=mesh)
```

Build the Stokes equations.

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> xVelocityEq = ImplicitDiffusionTerm(coeff=viscosity) - pressure.getGrad().dot([1,0])
>>> yVelocityEq = ImplicitDiffusionTerm(coeff=viscosity) - pressure.getGrad().dot([0,1])
```

In this example the SIMPLE algorithm is used to couple the pressure and momentum equations. Let us assume we have solved the discretized momentum equations using a guessed pressure field p^* to obtain a velocity field $\vec{u}^*$. We would like to somehow correct these initial fields to satisfy both the discretized momentum and continuity equations. We now try to correct these initial fields with a correction such that $\vec{u} = \vec{u}^* + \vec{u}'$ and $p = p^* + p'$, where $\vec{u}$ and p now satisfy the momentum and continuity equations. Substituting the exact solution into the equations we obtain,

$$\nabla \mu \cdot \nabla \vec{u}' = \vec{p}'$$

and

$$\nabla \cdot \vec{u}^* + \nabla \cdot \vec{u}' = 0$$

We now use the discretized form of the equations to write the velocity correction in terms of the pressure correction. The discretized form of the above equation is,

$$a_P \vec{u}'_P = \sum_f a_A \vec{u}'_A - V_P (\nabla p')_P$$

where notation from section 3.4 is used. The SIMPLE algorithm drops the second term in the above equation to leave,

$$\vec{u}'_P = -\frac{V_P (\nabla p')_P}{a_P}$$

By substituting the above expression into the continuity equations we obtain the pressure correction equation,

$$\nabla \frac{V_P}{a_P} \cdot \nabla p' = \nabla \cdot \vec{u}^*$$

In the discretized version of the above equation V_P / a_P is approximated at the face by $A_f d_{AP} / (a_P)_f$. In FiPy the pressure correction equation can be written as,

```
>>> ap = CellVariable(mesh=mesh)
>>> coeff = mesh._getFaceAreas() * mesh._getCellDistances() / ap.getArithmeticFaceValue()
>>> pressureCorrectionEq = ImplicitDiffusionTerm(coeff=coeff) - velocity.getDivergence()
```

Set up the no-slip boundary conditions

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> bcs = (FixedValue(faces=mesh.getFacesLeft(), value=0),
...        FixedValue(faces=mesh.getFacesRight(), value=0),
...        FixedValue(faces=mesh.getFacesBottom(), value=0),)
>>> bcsX = bcs + (FixedValue(faces=mesh.getFacesTop(), value=1),)
>>> bcsY = bcs + (FixedValue(faces=mesh.getFacesTop(), value=0),)
```

Set up the viewers,

```
>>> if __name__ == '__main__':
...     from fipy.viewers import make
...     viewer = make(vars=(pressure, xVelocity, yVelocity, velocity))
```

Below, we iterate for a set number of sweeps. We use the `sweep()` method instead of `solve()` because we require the residual for output. We also use the `cacheMatrix()`, `getMatrix()`, `cacheRHSvector()` and `getRHSvector()` because both the matrix and RHS vector are required by the SIMPLE algorithm. Additionally, the `sweep()` method is passed an `underRelaxation` factor to relax the solution. This argument cannot be passed to `solve()`.

```
>>> for sweep in range(sweeps):
...
...         ## solve the Stokes equations to get starred values
...         xVelocityEq.cacheMatrix()
...         xres = xVelocityEq.sweep(var=xVelocity,
...                                  boundaryConditions=bcsX,
...                                  underRelaxation=velocityRelaxation)
...         xmat = xVelocityEq.getMatrix()
...
...         yres = yVelocityEq.sweep(var=yVelocity,
...                                  boundaryConditions=bcsY,
...                                  underRelaxation=velocityRelaxation)
...
...         ## update the ap coefficient from the matrix diagonal
...         ap[:] = -xmat.takeDiagonal()
...
...         ## update the face velocities based on starred values
...         velocity[:,0] = xVelocity.getArithmeticFaceValue()
...         velocity[:,1] = yVelocity.getArithmeticFaceValue()
...         for id in mesh.getExteriorFaces():
...             velocity[id,:] = 0.
...
...         ## solve the pressure correction equation
...         pressureCorrectionEq.cacheRHSvector()
...         pres = pressureCorrectionEq.sweep(var=pressureCorrection)
...         rhs = pressureCorrectionEq.getRHSvector()
...
```

```
...        ## update the pressure using the corrected value but hold one cell fixed
...        pressure.setValue(pressure + pressureRelaxation * \
...                                        (pressureCorrection - pressureCorrection[0]))
...        ## update the velocity using the corrected pressure
...        xVelocity.setValue(xVelocity - pressureCorrection.getGrad()[:,0] / \
...                                        ap * mesh.getCellVolumes())
...        yVelocity.setValue(yVelocity - pressureCorrection.getGrad()[:,1] / \
...                                        ap * mesh.getCellVolumes())
...
...        if __name__ == '__main__':
...            if sweep%1 == 0:
...                print 'sweep:',sweep,', x residual:',xres, \
...                                    ', y residual',yres, \
...                                    ', p residual:',pres, \
...                                    ', continuity:',max(abs(rhs))
...
...                viewer.plot()
```
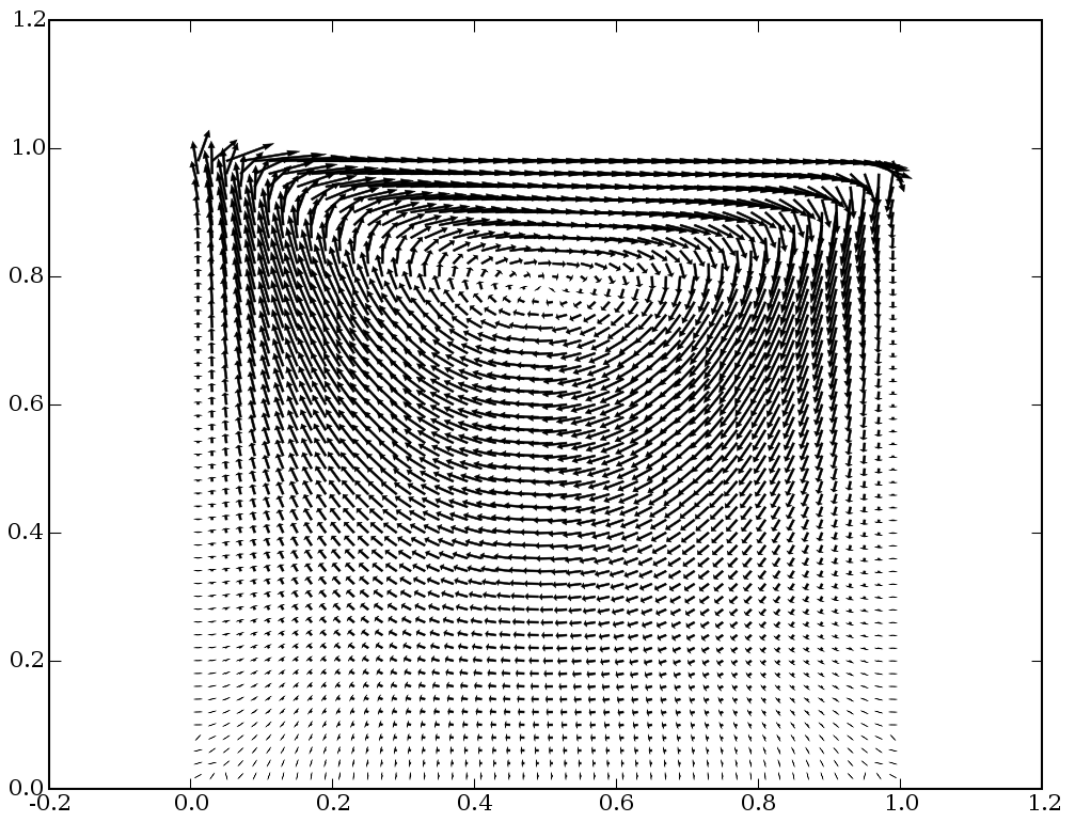

Test values in the last cell.

```
>>> from fipy.tools import numerix
>>> numerix.allclose(pressure[-1], 145.233883763)
1
>>> numerix.allclose(xVelocity[-1], 0.24964673696)
1
>>> numerix.allclose(yVelocity[-1], -0.164498041783)
1
```

Chapter 13

Converting from **FiPy** 0.1 to **FiPy** 1.0

13.1 Module `examples.update0_1to1_0`

 The syntax of FiPy scripts has changed considerably between version 0.1 and version 1.0. We have incremented the full version-number to stress that previous scripts are incompatible. We strongly believe that these changes are for the better, resulting in easier code to write and read as well as slightly improved efficiency, but we realize that this represents an inconvenience to our users that have already written scripts of their own. We will strive to avoid any such incompatible changes in the future.

All examples included with version 1.0 have been updated to use the new syntax, but any scripts you have written for FiPy 0.1 will need to be updated to work with FiPy 1.0. As a tutorial for updating your scripts, we will walk through updating the file `examples/convection/exponential1D/input.py` from FiPy 0.1. If you attempt to run that script with FiPy 1.0, the script will fail and you will see the errors shown below:

This example solves the steady-state convection-diffusion equation given by:

$$\nabla \cdot (D\nabla\phi + \vec{u}\phi) = 0$$

with coefficients $D = 1$ and $\vec{u} = (10, 0)$, or

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,0.)
```

We define a 1D mesh

```
>>> L = 10.
>>> nx = 1000
>>> ny = 1
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(L / nx, L / ny, nx, ny)
```

and impose the boundary conditions

$$\phi = \begin{cases} 0 & \text{at } x = 0, \\ 1 & \text{at } x = L, \end{cases}$$

or

```
>>> valueLeft = 0.
>>> valueRight = 1.
```

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> boundaryConditions = (
...        FixedValue(mesh.getFacesLeft(), valueLeft),
...        FixedValue(mesh.getFacesRight(), valueRight),
...        FixedFlux(mesh.getFacesTop(), 0.),
...        FixedFlux(mesh.getFacesBottom(), 0.)
...        )
```

The solution variable is initialized to `valueLeft`:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(
...        name = "concentration",
...        mesh = mesh,
...        value = valueLeft)
```

The `SteadyConvectionDiffusionScEquation` object is used to create the equation. It needs to be passed a convection term instantiator as follows:

```
>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> from fipy.solvers.linearCGSSolver import LinearCGSSolver
>>> from fipy.equations.stdyConvDiffScEquation import SteadyConvectionDiffusionScEquation
Traceback (most recent call last):
...
ImportError: No module named equations.stdyConvDiffScEquation
>>> eq = SteadyConvectionDiffusionScEquation(
...        var = var,
...        diffusionCoeff = diffCoeff,
...        convectionCoeff = convCoeff,
...        solver = LinearCGSSolver(tolerance = 1.e-15, steps = 2000),
...        convectionScheme = ExponentialConvectionTerm,
...        boundaryConditions = boundaryConditions
...        )
Traceback (most recent call last):
...
NameError: name 'SteadyConvectionDiffusionScEquation' is not defined
```

More details of the benefits and drawbacks of each type of convection term can be found in the numerical section of the manual. Essentially the `ExponentialConvectionTerm` and `PowerLawConvectionTerm` will both handle most types of convection diffusion cases with the `PowerLawConvectionTerm` being more efficient. We iterate to equilibrium

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eq,))
Traceback (most recent call last):
...
NameError: name 'eq' is not defined
>>> it.timestep()
Traceback (most recent call last):
...
NameError: name 'it' is not defined
```

and test the solution against the analytical result

$$\phi = \frac{1 - \exp(-u_x x/D)}{1 - \exp(-u_x L/D)}$$

or

```
>>> axis = 0
>>> x = mesh.getCellCenters()[:,axis]
>>> import Numeric
>>> CC = 1. - Numeric.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - Numeric.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
>>> Numeric.allclose(analyticalArray, var, rtol = 1e-10, atol = 1e-10)
0
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
Traceback (most recent call last):
...
ImportError: No module named grid2DGistViewer

...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

We see that a number of errors are thrown:

- `ImportError:  No module named equations.stdyConvDiffScEquation`

- `NameError:  name 'SteadyConvectionDiffusionScEquation' is not defined`

- `NameError:  name 'eq' is not defined`

- `NameError:  name 'it' is not defined`

- `ImportError:  No module named grid2DGistViewer`

As is usually the case with computer programming, many of these errors are caused by earlier errors. Let us update the script, section by section:

Although no error was generated by the use of `Grid2D`, FiPy 1.0 supports a true 1D mesh class, so we instantiate the mesh as

```
>>> L = 10.
>>> nx = 1000
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = L / nx, nx = nx)
```

The `Grid2D` class with `ny = 1` still works perfectly well for 1D problems, but the `Grid1D` class is slightly more efficient, and it makes the code clearer when a 1D geometry is actually desired.

Because the mesh is now 1D, we must update the convection coefficient vector to be 1D as well

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,)
```

The `FixedValue` boundary conditions at the left and right are unchanged, but a `Grid1D` mesh does not even have top and bottom faces:

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> boundaryConditions = (
...     FixedValue(mesh.getFacesLeft(), valueLeft),
...     FixedValue(mesh.getFacesRight(), valueRight))
```

The creation of the solution variable is unchanged:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(name = "concentration",
...                    mesh = mesh,
...                    value = valueLeft)
```

The biggest change between FiPy 0.1 and FiPy 1.0 is that `Equation` objects no longer exist at all. Instead, `Term` objects can be simply added, subtracted, and equated to assemble an equation. Where before the assembly of the equation occurred in the black-box of `SteadyConvectionDiffusionScEquation`, we now assemble it directly:

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> diffTerm = ImplicitDiffusionTerm(coeff = diffCoeff)

>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> eq = diffTerm + ExponentialConvectionTerm(coeff = convCoeff,
...                                            diffusionTerm = diffTerm)
```

One thing that `SteadyConvectionDiffusionScEquation` took care of automatically was that a `ConvectionTerm` must know about any `DiffusionTerm` in the equation in order to calculate a Peclet number. Now, the `DiffusionTerm` must be explicitly passed to the `ConvectionTerm` in the `diffusionTerm` parameter.

The `Iterator` class still exists, but it is no longer necessary. Instead, the solution to an implicit steady-state problem like this can simply be obtained by telling the equation to solve itself (with an appropriate `solver` if desired, although the default `LinearPCGSolver` is usually suitable):

```
>>> from fipy.solvers.linearCGSSolver import LinearCGSSolver
>>> eq.solve(var = var,
...          solver = LinearCGSSolver(tolerance = 1.e-15, steps = 2000),
...          boundaryConditions = boundaryConditions)
```

Note

In version 0.1, the Equation object had to be told about the `Variable`, `Solver`, and `BoundaryCondition` objects when it was created (and it, in turn, passed much of this information to the `Term` objects in order to create them). In version 1.0, the `Term` objects (and the equation assembled from them) are abstract. The `Variable`, `Solver`, and `BoundaryCondition` objects are only needed by the `solve()` method (and, in fact, the same equation could be used to solve different variables, with different solvers, subject to different boundary conditions, if desired).

The analytical solution is unchanged, and we can test as before

```
>>> Numeric.allclose(analyticalArray, var, rtol = 1e-10, atol = 1e-10)
1
```

or we can use the slightly simpler syntax

```
>>> print var.allclose(analyticalArray, rtol = 1e-10, atol = 1e-10)
1
```

The `ImportError: No module named grid2DGistViewer` results because the `Viewer` classes have been moved and renamed. This error could be resolved by changing the `import` statement appropriately:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.gistViewer.gist1DViewer import Gist1DViewer
...     viewer = Gist1DViewer(vars = var)
...     viewer.plot()
```

Instead, rather than instantiating a particular `Viewer` (which you can still do, if you desire), a generic "factory" method will return a `Viewer` appropriate for the supplied `Variable` object(s):

```
>>> if __name__ == '__main__':
...     import fipy.viewers
...     viewer = fipy.viewers.make(vars = var)
...     viewer.plot()
```

Please do not hesitate to contact us if this example does not help you convert your existing scripts to FiPy 1.0.

Bibliography

[1] The Python Programming Language, URL http://www.python.org/. 9

[2] W. J. Boettinger, J. A. Warren, C. Beckermann, and A. Karma, "Phase-field simulation of solidification". *Annual Review of Materials Research*, **32**, (2002) 163–194, URL http://arjournals.annualreviews.org/doi/abs/10.1146/annurev.matsci.32.101901.155803. 9, 81, 89

[3] L. Q. Chen, "Phase-field mdoels for microstructure evolution". *Annual Review of Materials Research*, **32**, (2002) 113–140, URL http://arjournals.annualreviews.org/doi/pdf/10.1146/annurev.matsci.32.112001.132041. 9, 81

[4] G. B. McFadden, "Phase-field models of solidification". *Contemporary Mathematics*, **306**, (2002) 107–145. 9, 81, 89

[5] D. Josell, D. Wheeler, W. H. Huber, and T. P. Moffat, "Superconformal Electrodeposition in Submicron Features". *Physical Review Letters*, **87**(1), (2001) 016102, URL http://link.aps.org/abstract/PRL/v87/e016102. 9, 121

[6] FiPy mailing list, URL http://www.ctcms.nist.gov/fipy/mail.html. 10, 44

[7] FiPy bug tracker, URL http://matdl-osi.org/fipy/report. 10, 44

[8] Greg Ward, *Installing Python Modules*. URL http://docs.python.org/inst/. 14

[9] Enthought Edition of Python for Windows and Linux, URL https://www.enthought.com/python/. 14

[10] Enthought Edition of Python for Mac OS X, URL http://old.scipy.org/wikis/featurerequests/MacEnthon. 14

[11] Python download page, URL http://www.python.org/download/. 15

[12] Matplotlib download page, URL http://sourceforge.net/project/showfiles.php?group_id=80706&package_id=82474. 16

[13] SciPy download page, URL http://sourceforge.net/project/showfiles.php?group_id=27747&package_id=19531&release_id=389739. 17

[14] FiPy download page, URL http://www.ctcms.nist.gov/fipy/download/. 18

[15] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato, *Version Control with Subversion*. O'Reilly Media, 2004, URL http://svnbook.red-bean.com. 18, 22

[16] Daniel Wheeler, Jonathan E. Guyer, and James A. Warren, *A Finite Volume PDE Solver Using Python*. URL http://www.ctcms.nist.gov/fipy/download/fipy.pdf. 18

[17] Guido van Rossum, *Python Tutorial*. URL http://docs.python.org/tut/. 19

[18] Mark Pilgrim, *Dive Into Python*. Apress, 2004, ISBN 1590593561, URL http://diveintopython.org. 19

[19] T. N. Croft, *Unstructured Mesh - Finite Volume Algorithms for Swirling, Turbulent Reacting Flows*. Ph.D. thesis, University of Greenwich, 1998, URL http://www.gre.ac.uk/~ct02/research/thesis/main.html. 23, 26

[20] S. V. Patanker, *Numerical Heat Transfer and Fluid Flow*. Taylor and Francis, 1980. 23

[21] H. K. Versteeg and W. Malalasekera, *An Introduction to Computational Fluid Dynamics*. Longman Scientific and Technical, 1995. 23

[22] C. Mattiussi, "An analysis of finite volume, finite element, and finite difference methods using some concepts from algebraic topology". *Journal of Computational Physics*, **133**, (1997) 289–309, URL http://lis.epfl.ch/publications/JCP1997.pdf. 23

[23] John W. Cahn and John E. Hilliard, "Free energy of a nonuniform system. I. Interfacial free energy". *Journal of Computational Physics*, **28**(2), (1958) 258–267. 24

[24] John W. Cahn, "Free energy of a nonuniform system. II. Thermodynamic basis". *Journal of Computational Physics*, **30**(5), (1959) 1121–1124. 24

[25] John W. Cahn and John E. Hilliard, "Free energy of a nonuniform system. III. Nucleation in a two-component incompressible fluid". *Journal of Computational Physics*, **31**(3), (1959) 688–699. 24

[26] James A. Warren, Ryo Kobayashi, Alexander E. Lobkovsky, and W. Craig Carter, "Extending Phase Field Models of Solidification to Polycrystalline Materials". *Acta Materialia*, **51**(20), (2003) 6035–6058, URL http://dx.doi.org/10.1016/S1359-6454(03)00388-4. 35, 104, 107

[27] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C: the Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1999. 55

[28] J. A. Warren and W. J. Boettinger, "Prediction of Dendritic Growth and Microsegregation in a Binary Alloy using the Phase Field Method". *Acta Metallurgica et Materialia*, **43**(2), (1995) 689–703. 86

[29] A. A. Wheeler, W. J. Boettinger, and G. B. McFadden, "Phase-field model for isothermal phase transitions in binary alloys". *Physical Review A*, **45**(10), (1992) 7424–7439. 89

[30] J. E. Guyer, W. J. Boettinger, J. A. Warren, and G. B. McFadden, "Phase field modeling of electrochemistry I: Equilibrium". *Physical Review E*, **69**, (2004) 021603, cond-mat/0308173, URL http://link.aps.org/abstract/PRE/v69/e021603. 99

[31] J. E. Guyer, W. J. Boettinger, J. A. Warren, and G. B. McFadden, "Phase field modeling of electrochemistry II: Kinetics". *Physical Review E*, **69**, (2004) 021604, cond-mat/0308179, URL http://link.aps.org/abstract/PRE/v69/e021604. 99

[32] J. A. Sethian, *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1996. 115

[33] T. P. Moffat, D. Wheeler, and D. Josell, "Superfilling and the Curvature Enhanced Accelerator Coverage Mechanism". *The Electrochemical Society, Interface*, **13**(4), (2004) 46–52, URL http://www.electrochem.org/publications/interface/winter2004/IF12-04-Pg46.pdf. 121

[34] D. Wheeler, D. Josell, and T. P. Moffat, "Modeling Superconformal Electrodeposition Using The Level Set Method". *Journal of The Electrochemical Society*, **150**(5), (2003) C302–C310. 122

[35] D. Josell, D. Wheeler, and T. P. Moffat, "Gold superfill in submicrometer trenches: Experiment and prediction". *Journal of The Electrochemical Society*, **153**(1), (2006) C11–C18. 126

[36] T. P. Moffat, D. Wheeler, S. K. Kim, and D. Josell, "Curvature enhanced adsorbate coverage model for electrodeposition". *Journal of The Electrochemical Society*, **153**(2), (2006) C127–C132. 127

[37] J. H. Ferziger and M. Perić, *Computational Methods for Fluid Dynamics*. Springer, 1996. 143

Index

Contributors

Jon Guyer is a member of the research staff of the Metallurgy Division in the Materials Science and Engineering Laboratory at the National Institute of Standards and Technology. Jon's computational interests are in object-oriented design and in phase field modeling of electrochemistry.

Daniel Wheeler is a caveman. Daniel's interests are in numerical modeling, finite volume techniques, and level set treatments.

Jim Warren is the group leader of the Themodynamics and Kinetics group in the Metallurgy Division and Director of the Center for Theoretical and Computational Materials Science of the Materials Science and Engineering Laboratory at the National Institute of Standards and Technology. Jim is interested in a variety of problems, including the phase field modeling of solidification, polycrystalline solids, and the electrochemical interface.

Alex Mont is a student at Montgomery Blair High School. Alex developed the *PyxViewer* and the *Gmsh* import and export modules.